CL2006 Operating Systems Lab Manual
ver 3.0

Saad Ahmad, *BS*
Inst., Dept. of Computer Science
National University of Computer & Emerging Sciences

Operating Systems Lab Manual
Saad Ahmad
Instructor, Dept. of Computer Science
National University of Computer & Emerging Sciences
160 Industrial Estate, Peshawar, KPK, Pakistan

Version History
1.0 Spring 2009: Installation, exercises spread over 11 lab sessions {**Omar Usman Khan**}
1.1 Summer 2009: New set of exercises only spread over 11 lab sessions {**Omar Usman Khan**}
2.0 Spring 2010: Compilation of all previous labs in book format with minor updates & corrections {**Omar Usman Khan**}
3.0 Spring 2025: Add some chapters with major updates & corrections {**Saad Ahmad**}

This manual is customized for course CL2006 Operating Systems taught at the National University of Computer & Emerging Sciences, Peshawar, Pakistan. The lab portion of this course provides cover for 1 credit hour. Evaluation for this lab is at discretion of your instructor and/or teacher.

The manual can also be used as a learning tool for other operating system courses both at undergraduate or graduate level. Each chapter in this manual is intended to serve as a separate lab, although instructors should note that some chapter's require more than one lab sessions to complete. The manual is based on the Ubuntu Linux Operating Systems version 2022.

Contents

Part I

Introduction & Installation

Chapter 1

Turning on your PC

1.1 How PC turns on

When you press the power button of your PC, the first thing which starts is called Firmware. A firmware is low level software which provides an interface between hardware and Operating System. The firmware is responsible for initializing the hardware, loading OS and transferring hardware control to OS. Firmware is stored permanently in ROM (Read Only Memory). Famous Computer firmwares:

- BIOS (Basic Input Output System) (legacy mode)
- UEFI (Unified Extensible Firmware Interface)

1.2 Firmware: BIOS and UEFI

Firmware is the piece of software that acts as an interface between the hardware (motherboard) and the operating system (OS). The difference between Unified Extensible Firmware Interface (UEFI) boot and legacy boot is the process that the firmware uses to find the boot target.

Legacy boot is the boot process used by basic input/output system (BIOS) firmware. BIOS was the first popular firmware for desktop PCs introduced in 1975 by IBM for its Control Program for Microcomputers (CP/M) OS. Even though it is still widely present, computers have evolved tremendously and BIOS is unable to provide advanced features of modern hardware. The firmware maintains a list of installed storage devices that may be bootable (floppy disk drives, hard disk drives, optical disk drives, tape drives, etc.) and enumerates them in a configurable order of priority. Once the power-on self-test (POST) procedure has completed, the firmware loads the first sector of each of the storage targets into memory and scans it for a valid master boot record (MBR). If a valid MBR is found, the firmware passes execution to the boot loader code found in the MBR, which allows the user to select a partition to boot from. If one is not found, it proceeds to the next device in the boot order. If no MBR is found at all, the user is presented with the famous, "Please insert system disk yadda yadda yadda," message.

UEFI boot is the successor to BIOS. UEFI uses the globally unique identifier (GUID) partition table (GPT) whereas BIOS uses the master boot record (MBR) partitioning scheme. GPT and MBR are both formats specifying physical partitioning information on the hard disk. The firmwaremaintains a list of valid boot volumes called EFI Service Partitions. During the POST procedure, UEFI firmware scans the bootable storage devices that are connected to the system for a valid GPT. Unlike a MBR, a GPT does not contain a boot loader. The firmware itself scans the GPTs to find an EFI Service Partition to boot from. If no EFI bootable partition is found, the firmware can fall back on the Legacy Boot method. If both UEFI boot and Legacy boot fail, the user is presented with the famous, "Please insert system disk yadda yadda yadda," message.

Below is the primary difference between both boot processes:

- Max partition size in MBR is ~2 TB, whereas in UEFI it is ~9 ZB
- MBR can have, at max, 4 primary partitions, whereas GPT can have 128.
- MBR can store only one bootloader, whereas GPT has a separate dedicated EFI system partition (ESP) for storing multiple bootloaders. This is very helpful if you have two or more operating systems which require different bootloaders.
- UEFI offers secure boot, which can prevent boot-time viruses from loading.

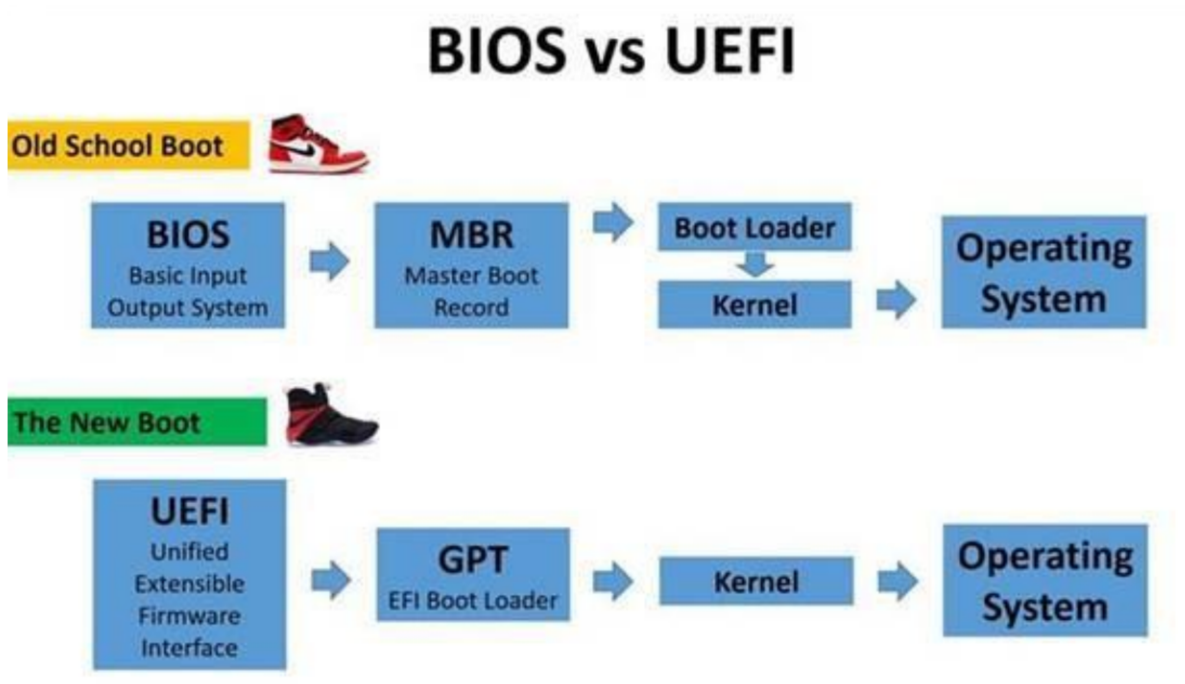
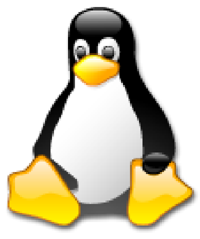


Figure 1.1

1.2.1 Introduction to Linux



Linux is an operating system which is a flavor of Unix. It is a multi-user and multi-tasking operating system. It was developed by Linus Torvalds at the University of Helsinki in Finland. The interface between Linux and it's users is known as the shell, i.e. user interacts with the Linux operating system through it. There may be two tasks to be performed by a shell. First, accepts commands from a user and second, interprets those commands. The core of the Linux OS is the kernel. As long as the system is operational, the kernel is running. The kernel is the part of the Linux Operating system which consist of routines, which interact with underlying hardware, and routines which include system call handling for process management, memory management, process scheduling, communication, file system, etc.

1.2.2 Ubuntu

Linux operating systems are presented in different flavours called distributions. One popular flavour of Linux is the Ubuntu Operating System. Other popular distributions are Arch, Manjaro, Red Hat, Fedora, Gentoo, Mint etc. Each distribution has its own set of specialties and functions. Ubuntu is a Debian-based distribution designed to be user-friendly and comes with a graphical user interface by default. It is widely used due to its ease of installation, robust support, and vast community resources.

Ubuntu users will use the following core steps to install their operating system via an ISO image or bootable USB:

1. Download Ubuntu ISO image.
2. Create a bootable USB drive.
3. Boot from the USB drive.
4. Follow the graphical installation process.
5. Configure user and system settings.
6. Install additional software or updates after installation.

1.3 Installation Steps

The following set of instructions/commands are customized for installing Ubuntu in a virtual or physical environment.

1.3.1 Preparing the Installation Medium

- Download the Ubuntu ISO image from the official website:

  `https://releases.ubuntu.com/jammy/`

- Create a bootable USB drive using a tool like *Rufus* (Windows) or *Startup Disk Creator* (Ubuntu):
  - Insert a USB drive (minimum 8 GB).
  - Launch *Rufus* or equivalent tool.
  - Select the downloaded Ubuntu ISO image.
  - Choose the USB drive and click *Start*.
- Once the bootable USB drive is created, safely eject it.

1.3.2 Booting into the Ubuntu Installer

- Insert the bootable USB drive into the target system.
- Restart the system and access the BIOS/UEFI settings (usually by pressing *F2*, *F12*, or *DEL* during boot).
- Change the boot order to prioritize the USB drive.
- Save the changes and reboot. The system will boot into the Ubuntu installer.

1.3.3 Installing Ubuntu

- Select the preferred language and click *Install Ubuntu*.
- Choose the keyboard layout and click *Continue*.
- Connect to a Wi-Fi network if necessary.
- Choose installation options:
  - *Normal Installation*: Includes a GUI, web browser, office software, and media players.
  - *Minimal Installation*: A lighter version with only essential software.
- Check the option to *Download updates while installing Ubuntu* (recommended).
- If needed, check the *Install third-party software for graphics and Wi-Fi hardware* option.
- Click *Continue*.

1.3.4 Partitioning the Disk

- Select the installation type:
  - *Erase disk and install Ubuntu*: Simplifies installation by using the entire disk.
  - *Something else*: Allows manual partitioning. **Recommended for dual boot**.
- If choosing manual partitioning, follow these steps:
  - Create a root (*/*) partition (minimum 20 GB, ext4 filesystem).
  - Create a swap partition (recommended size is 2 GB to 8 GB depending on system RAM).
  - Optionally, create a home (*/home*) partition for user data.
- Click *Install Now* and confirm changes.

1.3.5 Configuring User and System Settings

- Set your location to configure the system time zone.
- Enter your name, computer name, username, and password.
- Choose whether to log in automatically or require a password at login.
- Click *Continue* to start the installation.

1.3.6 Completing Installation

- Once the installation is complete, remove the USB drive when prompted.
- Click *Restart Now*.
- After rebooting, log in using the credentials you created during installation.

1.3.7 Post-Installation Steps

- Update the system:

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

- Install additional software as needed using the *Ubuntu Software Center* or *apt* package manager.

- Configure system settings such as display, sound, and network.

1.3.7.1 A Note on Ubuntu Directory Structure

- The Ubuntu file system follows the standard Linux directory structure:
  - */* (Root): The top-most directory.
  - */home*: User data directory.
  - */etc*: System configuration files.
  - */var*: Variable data files such as logs.
  - */usr*: Installed software and libraries.
  - */boot*: Bootloader and kernel-related files.

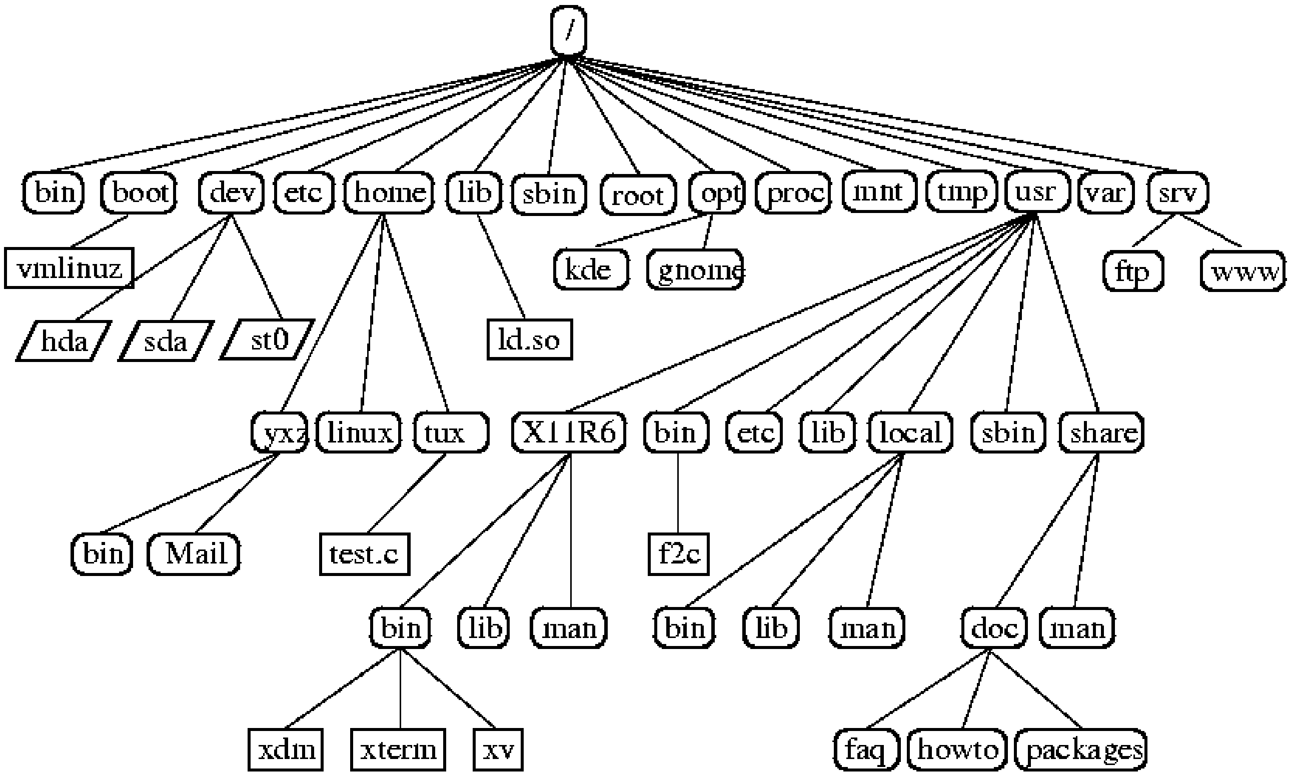


Figure 1.2

Chapter 2

Getting To Know Linux

2.1 Ubuntu's File System Structure

Ubuntu uses the Linux file system, which is based on a series of folders in the root directory. These folders contain important system files that cannot be modified unless you are running as the root user or use sudo. This restriction exists for both security and safety reasons; computer viruses will not be able to change the core system files, and ordinary users should not be able to accidentally damage anything vital.

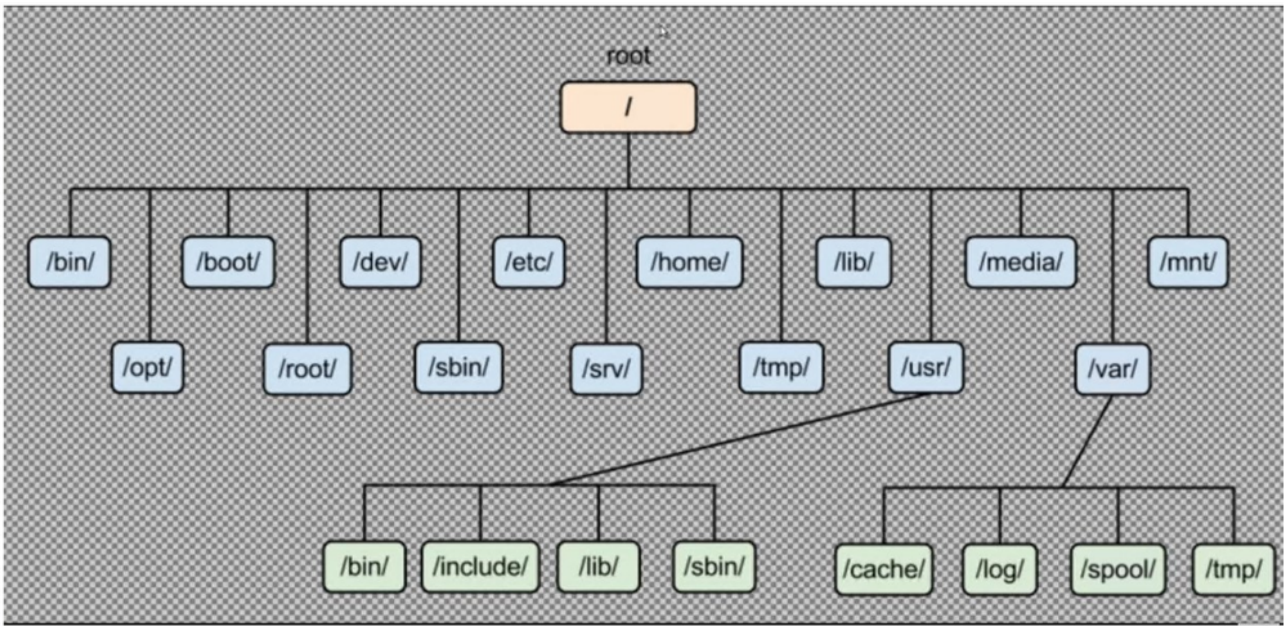


Figure 2.1

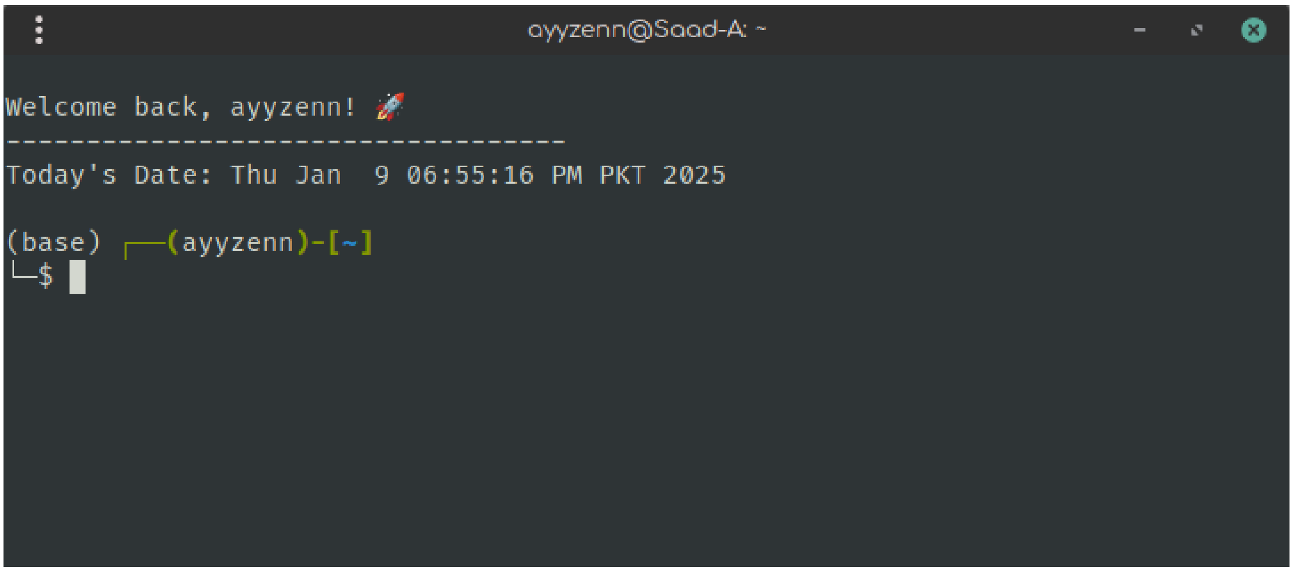
At the top of the hierarchy is the root directory which is denoted by /. The root directory contains all other directories and files on your system. Below the root directory are the following essential directories:

- **/bin** and **/sbin** Many essential system applications (equivalent to C:\Windows).
- **/etc** System-wide configuration files.
- **/home** Each user will have a subdirectory to store personal files (for example, **/home-/yourusername**) which is equivalent to **C:\Users** or **C:\Documents and Settings** in Microsoft Windows.

- **/lib** Library files, similar to .dll files on Windows.
- **/media** Removable media (cd-roms and usb drives) will be mounted in this directory.
- **/root** This contains the root user's files (not to be confused with the root directory).
- **/usr** Pronounced "user", it contains most program files (not to be confused with each user's home directory). This is equivalent to C:\Program Files in Microsoft Windows.
- **/var/log** Contains log files written by many applications.

2.2 Terminal

To fully harness the capabilities of Ubuntu, it is important to learn how to use the terminal. Most operating systems, including Ubuntu, provide two types of user interfaces: the graphical user interface (GUI) and the command-line interface (CLI). The GUI consists of the desktop, windows, menus, and toolbars that users interact with through clicks and gestures. In contrast, the CLI offers a more direct way to control the system through typed commands. Ubuntu's CLI is accessed via the terminal, which enables users to execute various commands to manage and customize the system. By default, Ubuntu uses the *bash* shell as its terminal interface, providing powerful tools for scripting, file management, and system administration.



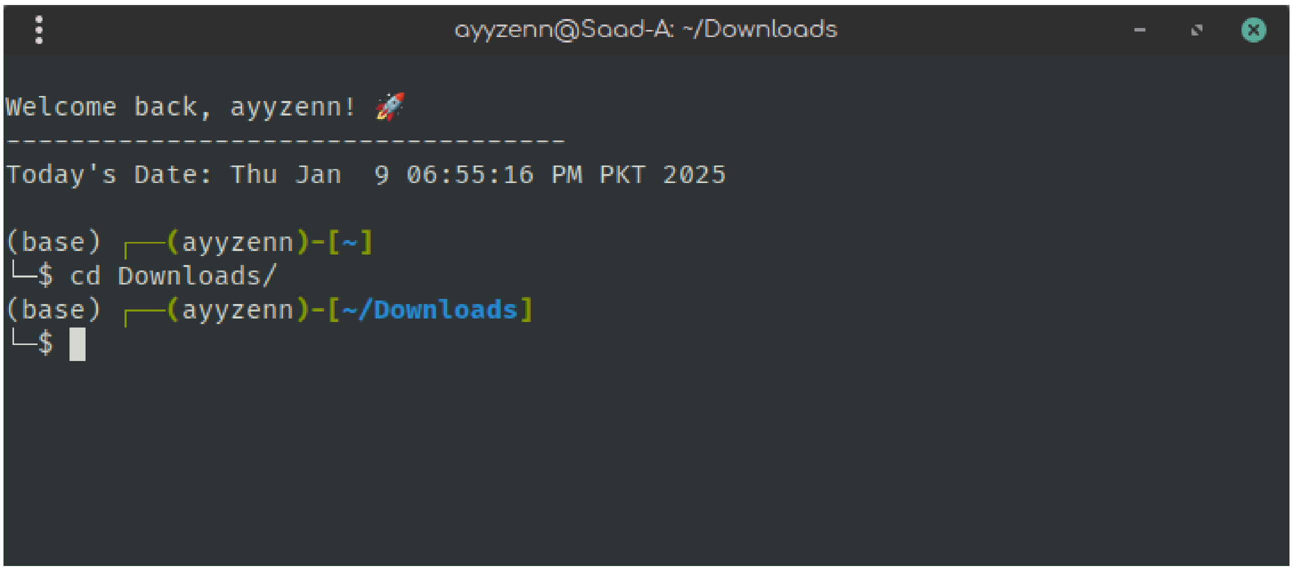
```
Welcome back, ayyzenn! 🚀
------------------------------------
Today's Date: Thu Jan  9 06:55:16 PM PKT 2025

(base) ┌──(ayyzenn)-[~]
└─$
```

Figure 2.2: Terminal Window

2.2.1 Why would you want to use the terminal?

The terminal provides access to what is called a shell. When you type a command in the terminal, the shell interprets the command and executes the desired action. All terminal commands follow a similar structure: type a command, optionally add parameters (also called switches), and press Enter to execute the action. Parameters modify how the command behaves and are usually appended to the command, often in the form of `-h` or `--help`. Adding `--help` to most commands displays a brief description of the command along with a list of possible parameters. Typically, the terminal outputs a confirmation when an action completes successfully, although this depends on the command. For instance, using the `cd` command to change the current directory updates the prompt but does not produce additional output, as illustrated in the figure below.



```
Welcome back, ayyzenn! 🚀
-----------------------------------
Today's Date: Thu Jan  9 06:55:16 PM PKT 2025

(base) ┌──(ayyzenn)-[~]
└─$ cd Downloads/
(base) ┌──(ayyzenn)-[~/Downloads]
└─$
```

Figure 2.3

2.3 Commands

A command is a request from a programmer, an operator, or a user to Linux operating system asking that a specific function be performed. For e.g., a request to list all files in your current directory will be the command **ls**.

2.3.1 Syntax of Commands

The general way commands are entered in Linux is as such:

```
command -option(s) argument(s)
```

Here,

- A command tells the operating system what to do (what action to be performed, copy a file, display a date etc.)
- Option(s) tells the way of action to be performed. For example, ls command displays directory contents, and –r option tells the way in which the directory should be displayed. Here –r displays directory contents in reverse (alphabetically) order.
- Argument tells that on what objects (file, directory, devices, etc.) the command and its arguments are applied. For example if we need to display all files starting with alphabet a, you will give "ls a*" and press enter.

Note: Make sure you don't forget that there is always a space between the command, the options, and the arguments.

2.4 More about Shell

2.4.1 The Asterisk *

The asterisk (*) symbol is a wildcard that can be used in various contexts. For example:

- It can denote *everything*. For example, in Linux, typing `rm *` will delete all files in the current directory.
- It can be used as a filter. For instance, typing `ls ab*` will list all files and folders that start with `ab`.

2.4.2 Case Sensitivity

Linux commands are case-sensitive. All standard Linux commands are in lowercase letters. For example:

- Typing `ls` will list the directory contents.
- Typing `LS`, `lS`, or `Ls` will result in a command syntax error.

2.4.3 Auto-Completion

Auto-Completion is a shortcut feature to quickly enter long commands or ones you may not fully remember. To practice:

- Type a key letter, e.g., `f`, and press `TAB`. You will see a list of commands starting with `f`.
- Add more letters, e.g., `fd`, and press `TAB` again. The list narrows down.

You can also use auto-completion for directories. For example, if you want to access the home directory of a user named `abcdefghijklmnopqrstuvwxyz`:

- From the root directory, type `cd /home/a` and press `TAB`. The rest of the name will auto-complete, saving you time.

2.4.4 Redirection

Redirection uses `>` and `<` symbols to capture output or input. The types of redirection are:

- `>`: Redirect output to a file.
- `1>`: Same as `>`.
- `2>`: Redirect error output to a file.
- `<`: Redirect input from a file.
- `> >`: Apend to a file.

Example Commands:

```
$ cd
$ ls
$ touch newfile
$ ls
$ ls > newfile
$ cat < newfile
$ ls -lh
$ ls -lh 1> newfile
$ cat < newfile
$ lsot
$ lsot 2> newfile
$ cat < newfile
$ rm newfile
```

2.5 Practicing Commands

Some of the most commonly used commands are listed below. Try practicing each to see what they do.

`ls` Print the list of directory contents of the current directory (or another directory if the full path is specified).

`cd` Change the current directory.

`mkdir` Create a new directory.

`rmdir` Remove an empty directory.

`cat` View or write contents of a file.

`cp` Copy a file from one location to another.

`mv` Move a file from one location to another.

`rm` Remove files or directories.

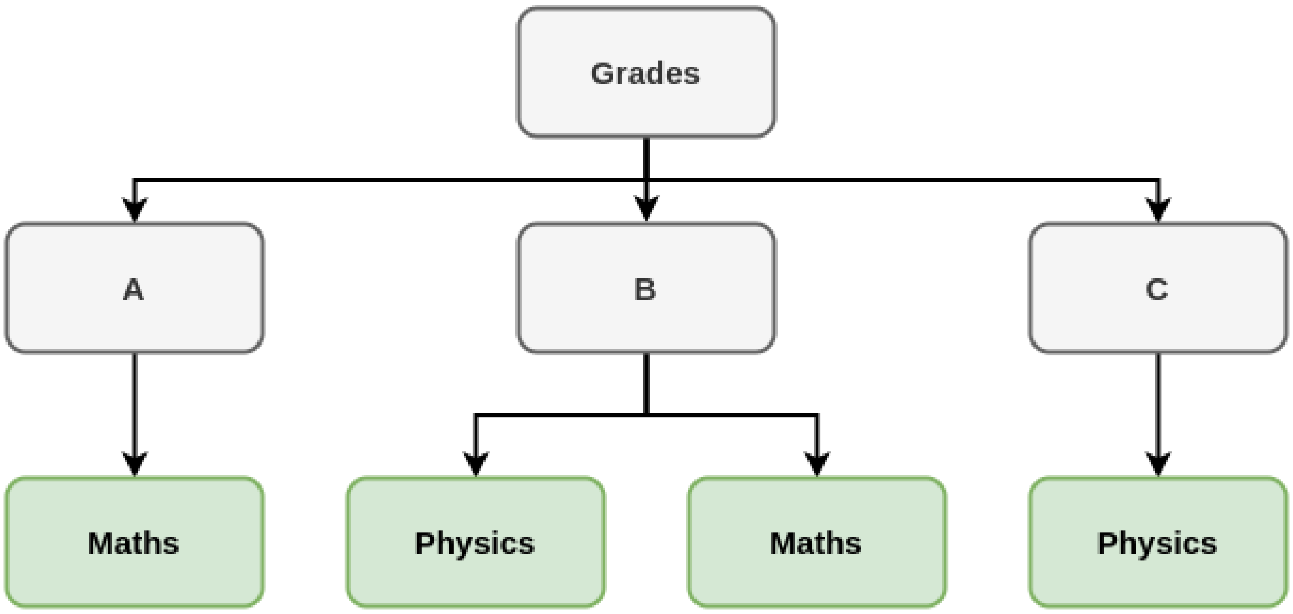
Example Commands:

```
$ mkdir temporary
$ cd temporary
temporary$ ls
temporary$ cat > newfile
Type any text and press CTRL+D
temporary$ cat newfile
temporary$ mkdir another
temporary$ cp newfile another/newest
temporary$ cp newfile newester
temporary$ cd another
another$ ls
another$ ls -a
another$ ls -l
another$ ls -lh
another$ cp newest newestest
another$ cat newestest
another$ cd ..
temporary$ mv newester another/newester
temporary$ ls
temporary$ ls another/n*
temporary$ cd ..
$ rm temporary
$ rm temporary/*
$ rm temporary
$ rm temporary -r -f
```

Easy? Okay, try and attempt the following exercise.

2.5.1 Exercise 1

Implement the following directory tree.



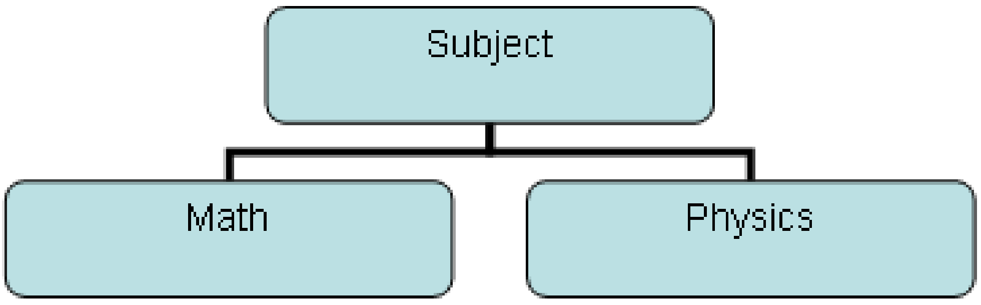
The boxes in white are directories. The boxes in light green are files. Each file should contain a random marks of your liking.

Once done, delete all the files/directories that you have created.

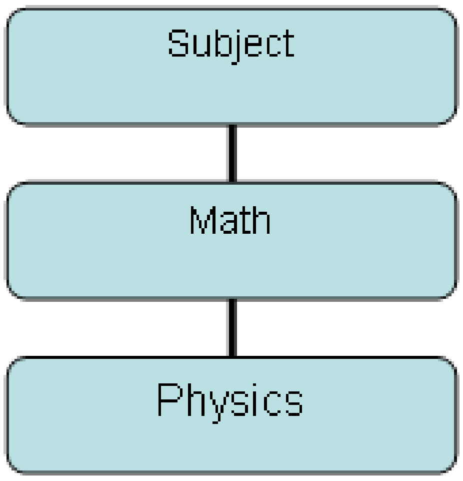
Count the total number of commands you entered to do this job. I managed using 6 commands.

2.5.2 Exercise 2

Consider the following directory tree:



- How are we going to change it using the “mv” command so that we get the directory tree as below:



- Change your directory so that your current directory is “Physics”. From here, change your directory in only one command such that your current directory becomes “subject”.
- From “subject”, issue only one command such that the directory “physics” is deleted.

2.6 Text Editor

A text editor is a software where you can enter text in its native format and save it to file. A word processor is a software where you can take text and process its appearance, format, spell-check, paragraph settings, etc. Linux has a number of text editors (both graphical and command-line based). The one which you will use most commonly is the nano text editor.

2.6.1 NANO Editor

There are many text editors available for Linux. At the moment you will have access to the *nano* editor. Nano is an advanced text editor provided by GNU. Simply typing *nano* on the shell will give you the editor as shown in Figure-2.4.

```
nano
```

You may also start your nano by explicitly mentioning the file you want to work on. This file may be already existing or you may be creating a new one.

```
nano <myFile>
```

Near the end of your screen you will see a list of shortcuts. The ones which you should get yourself familiar with are as such:

CTRL+X for exit

CTRL+O for saving

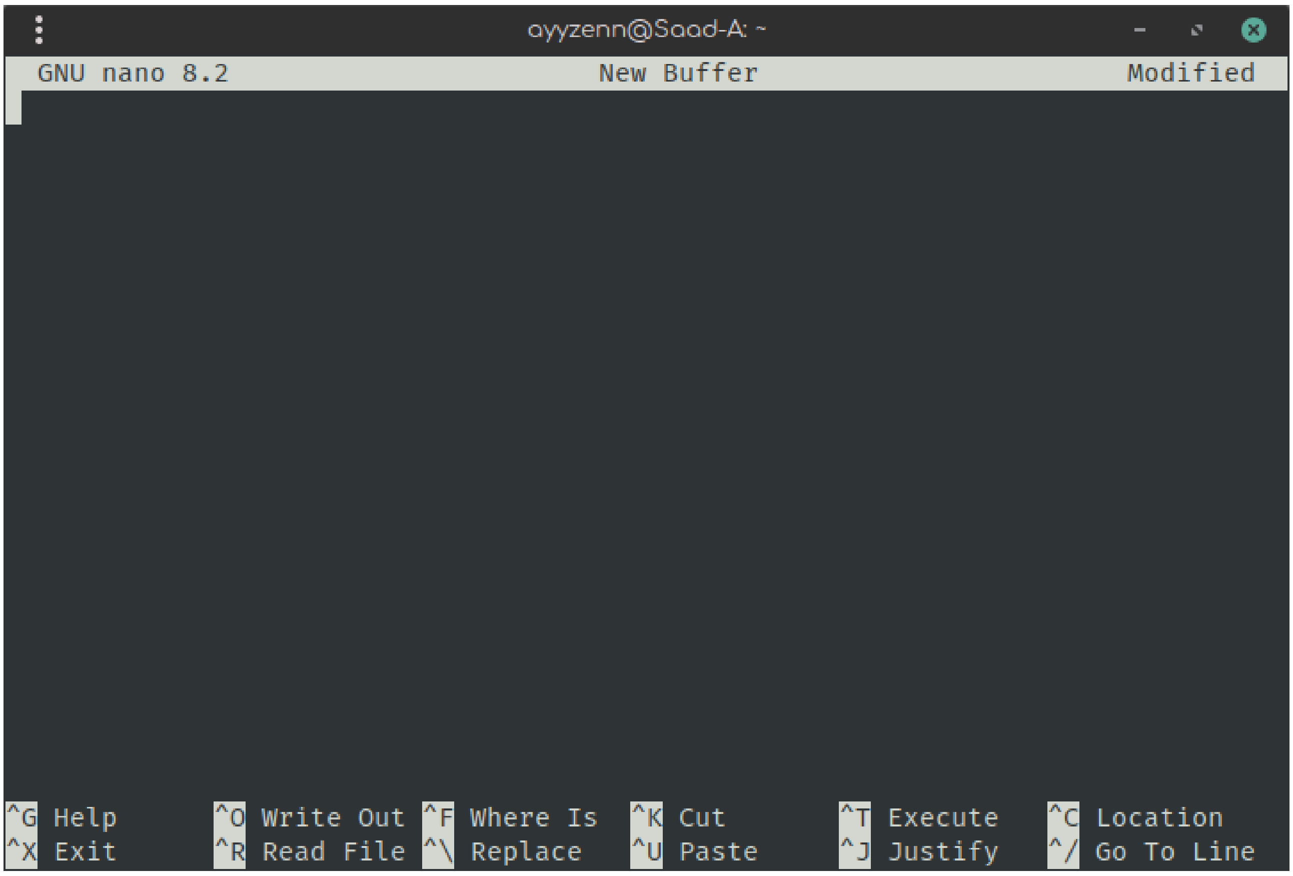
CTRL+W for searching

CTRL+K for cutting

CTRL+U for pasting

CTRL+C for displaying cursor position

Other commands are listed at the bottom of the text-editor window.



```
ayyzenn@Saad-A: ~
 GNU nano 8.2                  New Buffer                          Modified

^G Help      ^O Write Out ^F Where Is  ^K Cut       ^T Execute   ^C Location
^X Exit      ^R Read File ^\ Replace   ^U Paste     ^J Justify   ^/ Go To Line
```

Figure 2.4: Nano Text Editor

2.6.2 VIM Editor

Vim (Vi IMproved) is a highly configurable text editor built for efficient text editing. It is an improved version of the Unix-based ‘vi‘ editor. To open Vim, simply type the following on the shell:

```
vim
```

You can also start Vim by specifying a file you want to edit. If the file does not exist, Vim will create a new one:

```
vim <myFile>
```

Vim operates in different modes, and some of the basic modes include:

Normal Mode This is the default mode where commands can be issued.

Insert Mode For entering or editing text.

Command Mode Used to perform various actions like saving or exiting.

Key commands to familiarize yourself with include:

`ESC` Switch to Normal Mode.

`:w` Save the file.

`:q` Quit Vim.

`:wq` Save and quit Vim.

`:q!` Quit without saving changes.

`i` Enter Insert Mode.

`/` Search for text.

`dd` Delete a line.

`yy` Copy (yank) a line.

`p` Paste a copied line.

`u` Undo the last change.

`CTRL+R` Redo the last undone change.

Below is a typical example of how to use Vim:

```
vim myFile.txt
# Press "i" to enter Insert Mode and edit your file
# Press "ESC" to return to Normal Mode
# Type ":wq" to save and exit
```

```
NORMAL  [No Name]                               [unix]  100%  ln :0/1≡%  :1
```

Figure 2.5: VIM Text Editor

2.7 Links

Links are the equivalent of Shortcuts in Windows. A link in UNIX/Linux is a pointer to a file or directory, similar to pointers in programming languages. Links allow more than one file name to refer to the same file elsewhere, serving as shortcuts for access.

There are two types of links:

- Soft Links (Symbolic Links)
- Hard Links

These links behave differently when the source of the link (the target) is moved or removed. Symbolic links merely store the path name of the target, while hard links always point to the source even if it is moved or deleted.

For example, if you create a hard link to a file `a.txt` and then delete the file, you can still access the file using the hard link. However, if you create a soft link to the file and then delete the file, the soft link becomes dangling and no longer works.

2.7.1 Hard Links

- Each hard-linked file shares the same Inode value as the original file, referencing the same physical file location.
- Hard links remain linked even if the original or linked files are moved within the same file system.
- `ls -l` displays the number of links in the link column.
- Hard links store the actual file contents and do not become dangling if the original file is deleted.
- Changing the file name of the original file does not affect the hard links.
- Hard links cannot be created for directories or files across different file systems.
- The size of a hard-linked file is the same as the original file, and changes to any linked file are reflected in all hard links.

Command to create a hard link:

```
ln [original filename] [link name]
```

2.7.2 Soft Links (Symbolic Links)

- A soft link is similar to the shortcut feature in Windows. It has a separate Inode value that points to the original file.
- Changes to data in either the original file or the soft link are reflected in the other.
- Soft links can link across different file systems.
- If the original file is moved or deleted, the soft link becomes a dangling link (non-functional).
- Soft links can point to directories.

- `ls -l` displays soft links with an `l` in the first column, showing the path to the original file.
- The size of a soft link is equal to the name length of the original file.

Command to create a soft link:

```
ln -s [original filename] [link name]
```

2.7.3 Practice Commands

To practice, try out the following commands:

```
# ls
# touch blankfile
# mkdir blankdir
# ls
# ln blankdir dirpointer
# ln -s blankdir dirpointer
# ln blankfile filepointer
# cat < blankfile
# cat < filepointer
# echo "Hello dear" > filepointer
# cat < blankfile
# cat < filepointer
# ls
# ls -lh
```

Look carefully at the contents of both `filepointer` and `dirpointer`. Changing one will automatically change the other. Also, observe how the directory pointers appear using `ls -lh`.

2.8 Manuals

The most important thing to get yourself familiar with in linux is the *manuals*.

```
man
```

Usually a user has to specify a manual page. A manual page is usually listed by it's program name, command name, or system call name. For example:

```
man ls
man cp
man man
man fork
```

- To navigate, use the ↑or ↓ arrow keys.
- To search, type / (slash), followed by your search string, and enter
- To exit, type q

2.9 More | Less

Sometimes a user may give a command which generates so much output that it scrolls very fast. As a result, the top information simply flows out of the screen whereas only the low-end information is visible. For example, use the following command

```
ls /usr/bin -lh
```

We can view the lost information using the more or less commands. Try both of them first.

```
ls /usr/bin -lh | more
```

To quit, press **q**.

```
ls /usr/bin -lh | less
```

The syntax of both above commands are such that we are specifying two commands in one go. The first command starts with **ls**, the second command starts with **more** or **less**. Both these commands are **joined** by the pipe symbol |.

With more, you are able to browse **down** the display 1-page at a time using spacebar.

With less, you are able to browse **up** and **down** the display 1-line at a time using up and down arrow keys.

An alternative is using output redirection that you have covered in section-2.4.4. Usage would be as such:

```
ls /usr/bin -lh > temp.txt
nano temp.txt
```

2.10 Searching

By default, searching for files or directories is performed using the *find* command. Another powerful search program is *locate*.

2.10.1 Using Find

The syntax of find command is as such:

```
find <where> -name <what>
```

So, if I want to find all pdf files in / directory, I will give:

```
find / -name *.pdf
```

2.10.2 Using Locate

The locate program uses a database to search for files. The program gives results much more quickly than find command. The downside to updating the database. Updating is done using updatedb command. The format of command is also simple as compared to find.

```
sudo apt update && sudo apt install plocate -y
sudo updatedb
locate *.pdf
```

2.11 GNU Compiler Collection

Prorams written in C on linux are compiled using the gcc compiler. Programs written in C++ are compiled using the g++ compiler. Both these compilers are provided by GNU under the label GCC.

Any C program written for linux should have the extension of .c (e.g., hello.c), whereas a C++ program should have an extension of .cpp (e.g., hello.cpp). When compiling, the general syntax of command is as follows:

```
gcc first.c [-o first]
g++ first.cpp [-o first]
```

A breakdown is as such:

gcc|g++ is the compiler itself

first.c|first.cpp will be the filename with extension of your source code.

[ ] Anything inside this square brackets is an optional arguments. Do not write the square brackets themselves in your command.

-o is the output filename. It is the argument for making an executable file with the name you specify. Without this, the source code will be compiled into an executable file named *a.out*

first is the name of executable file which you pass to -o argument.

So considering that we have a source code with the name first.c, and compile it with the above command, we can execute it using:

```
./first
```

Where ./ specifies the current directory, and first is the name of executable file which you passed as an argument to the compiler. Try it out using the following code:

```
#include <stdio.h>
main() {
   printf("hello, world\n");
}
```

2.11.1 Exercise

- Write your very first C program for linux. Name it as *myFirst.c.* Your program should prompt the user for his name, and then should display the name on the shell. Tip: Use the printf() and the scanf() calls for this purpose & remember to check their respective man pages as well (man printf & man scanf)
- To run a linux command from your program, we can use the system() function. Use the following inside your myFirst.c program and see it's output.

  ```
  system("ls");
  ```

- Using the system() function, do the following:
  - Create a directory with the name test in your current directory.
  - Then display the contents of that directory
  - Finally erase the test directory

2.12 File Permissions

All files and directories in Linux have an associated set of owner permissions that are used by the operating system to determine access. These permissions are grouped into *three sets of three bits.* The sets represent {owner, group, everyone else} whereas the bits represent {read, write, execute}. So overall, we have 9 permissions (See Table 2.1).

	Read	Write	Execute
Owner	1	1	1
Group	1	1	0
Others	0	0	1
	2^2	2^1	2^0

Table 2.1: Representation of File Permissions

If we look at the first row for Owner, we see three 1's. Which specify that the Owner has read, write, and execute permissions on a file or directory. Since all of the bits are in allow mode, we can add them up together to get $2^2 + 2^1 + 2^0 = 4 + 2 + 1 = 7$. Similarly, the second row for Group, i.e., users within the same group as that of the file owner, have read and write, but no execute permissions. This will be translated only as $2^2 + 2^1 = 4 + 2 = 6$. And lastly, every other user can only execute files and have no permission to read or write to them. This will be translated as $2^0 = 1$. So the file permissions would be *761.* To give the permission of 761 to a file, we would use the command:

```
chmod 761 <filename>
```

You can view the file permissions using the command:

```
ls -lh
```

2.12.1 Changing File Permissions with the `chmod` Command

The `chmod` command, which stands for "change mode," is used to set permissions (read, write, execute) on a file or directory for the owner, group, and others.

2.12.1.1 Syntax

```
chmod permissions filename
```

There are two ways to use the `chmod` command:

1. Absolute (Numeric) Mode
2. Symbolic Mode

2.12.1.2 1. Absolute (Numeric) Mode

In this mode, file permissions are represented as a three-digit octal number. The table below explains the numeric values for different permission types:

Number	**Permission Type**	**Symbol**
0	No Permission	—
1	Execute	–x
2	Write	-w-
3	Execute + Write	-wx
4	Read	r–
5	Read + Execute	r-x
6	Read + Write	rw-
7	Read + Write + Execute	rwx

Table 2.2: Permission Values in Absolute Mode

For example, changing a file's permissions to `764` using:

```
chmod 764 sample
```

`764` means:

- Owner can read, write, and execute (`rwx`).
- Group can read and write (`rw-`).
- Others can only read (`r-`).

2.12.1.3 2. Symbolic Mode

In symbolic mode, permissions are changed for specific owners (user, group, or others) using mathematical symbols. This method uses:

- `+` to add a permission
- `-` to remove a permission
- `=` to set permissions and override previous ones

The owners are denoted as:

- `u`: User/Owner
- `g`: Group
- `o`: Others
- `a`: All

Examples

```
chmod o+x sample.txt # Give execute permission to others
chmod g+w sample.txt # Give write permission to group
chmod u+rwx sample.txt # Give all permissions to the user
chmod ugo+rwx sample.txt # Give all permissions to everyone
chmod a-rwx sample.txt # Remove all permissions from everyone
```

2.12.2 Directory Permissions

Directory permissions work similarly to file permissions but are applied to directory operations. For example:

```
chmod u+x sample # Add execute permission to access the directory
chmod g+wrx sample # Give all permissions to group
chmod o-wrx sample # Remove all permissions from others
```

Changing directory permissions affects the ability to access, modify, or list its contents. Use these commands with caution to avoid accidental access issues.

2.13 File Ownership

A file's owner can be changed using the command:

```
chown <username> <filename>
```

Where, *<username>* would be the username of any user on your system, and *<filename>* is the target to which this setting is going to apply to. The `chown` command stands for "change owner."

- `ch` stands for "change."
- `own` stands for "owner."

2.13.1 Examples

```
sudo chown ayyzenn sample
```

This command changes the directory's current owner to `ayyzenn`. Verify the change using:

```
ls -l
```

Similarly, to change the owner of a file:

```
sudo chown ayyzenn file.txt
ls -l
```

2.13.2 Changing User and Group Ownership

If you want to change both the user and the group for a file or directory, use the following syntax:

```
chown user:group filename
```

For example:

```
sudo chown ayyzenn:ayyzenn sample.txt
ls -l
```

2.13.3 Handling Permission Issues

When you attempt to write to a file owned by another user, you might encounter an error:

```
cat > sample.txt # Output:  "bash:  test.txt:  Permission denied"
```

This happens because the current user does not own the file. You can resolve this by changing the ownership of the file:

```
sudo chown ayyzenn:ayyzenn sample.txt
ls -l
```

2.13.4 Changing Group Ownership Only

To change the group ownership of a file or directory, use the `chgrp` command, which stands for "change group."

```
sudo chgrp ayyzenn sample.txt
```

Part II

System Calls

Chapter 3

Processes

A system call is an interface between your program and the kernel. The linux kernel's job is to provide a variety of services to application programs and this is done using the provision of system calls.

3.1 Understanding Processes

The fundamental building block of each program is the process. A process is the name given to a program when it is loaded "for" the purpose of execution on the operating system.

To view a list of current processes in the system for a user, you may use the ps command.

```
ps
```

To view the complete list, you can adapt it to:

```
ps au
```

You will see plenty of columns as output. The columns you should be familiar with at this moment are underlined below:

- **User**: The owner of that process
- **PID**: The integer identifier
- **CPU**: Percent utilization of CPU
- **MEM**: Percent utilization of Memory space
- **VSZ**: Virtual Memory Size
- **RSS**: Non-swapped physical memory size
- **TTY**: Controlling Terminal
- **STAT**: The process states (D) Uninterruptible sleep (R) Running or ready (S) Interruptible sleep (T) Stopped (Z) Zombie (<) High Priority (N) Low Priority (s) Session leader, i.e., has child processes (l) multi-threaded (+) foreground
- **START**: When process has started
- **TIME**: Time running since

- **COMMAND**: The program/command used in this process.

The basic attribute of a process is it's ID (PID), and it's Parent ID (PPID). The system calls that can find the process ID, and the Parent Process ID are the getpid() and getppid() calls respectively. To use both of them, you will be required to include the *sys/types.h* and *unistd.h* C libraries.

The mere presence of the PPID means that there is a hierarchy of processes inside our operating system. Each process has a track of who are it's children, and each child process has a record of who is it's parent. The original process that is the grand-father of all processes is the *init* Process. You can view this hierarchy using the pstree command.

```
pstree
```

This command will show a list of all processes currently in the system in the form of a tree.

3.1.1 Exercise

With respect to the myfirst.c file that you created in last labs, write a code that is able to find out the following:

- The PID value for myfirst.c
- The PPID value for myfirst.c
- The process name from the PPID value. (Note: To find a process name from PPID, you would normally enter the following command on the shell, where 12345 is the ID of any process)

```
pstree -p | grep 12345
```

3.2 Process Lifecycle

Each process has to go through the following states during it's existence:

1. Creation
2. Running
3. Non-Running
   (a) Ready
   (b) Waiting
4. Termination

3.2.1 Creation States

About process creation concepts, understand and run the following code:

```
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
int main(void)
{
 int i;
 printf("Process PID %6d \t PPID %6d \n", getpid(), getppid());
 for (i = 0; i<1; ++i)
 {
   if (fork()==0)
   printf("Process PID %6d \t PPID %6d \n", getpid(), getppid());
 }
 return 0;
}
```

Q1 How many processes are created?

Q2 Increase the value in for loop from i<1 to i<2 (i.e., 2 iterations in the loop). Compile and run your program. How many processes does it show this time? Draw a tree hierarchy of processes that you just created as given in Figure-3.1.

Q3 Increase the value again to i<3 (i.e., 3 iterations). Compile and run your program. How many processes does it show? Draw a tree again.

Q4 For fun, increase the value yet again to 100. Compile and run. What is going to happen? Does your OS Crash? Does your Program Crash? Can you modify your code to count the total number of fork()s made?

Try and run the following code:

```
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
int main(void)
{
   fork(); printf("He\n");
   fork(); printf("Ha\n");
   fork(); printf("Ho\n");
}
```

Note that we are calling each He, Ha, and Ho only once. Yet that does not appear when the program is run.

Q5 Can a Ho be output before a He? Why?

Explanation

The Fork() system call simply creates a new process which is exact replica of parent process. It requires the *unistd.h* C library. It's usage is as such:

```
// Fork code 1
#include <unistd.h>
#include <???> // See Question 2
```

```
int main()
{
  int p;
  p = fork();
  printf("Job Done\n");
}
```

To understand this code, try to answer the following questions:

Question1 We have used p = fork(). Why not simply fork()? Check **man fork** for answer.

Question2 Check **man page** for printf. What library is used for this call? - man 3 printf

Question3 Run your program. Why is it that printf() is used only once, yet we see the output "Job Done" displaying twice on our screen.

Question4 Add the following statement to the end of your code and run it again. What output would you see?

```
printf("Value of P is %d\n", p);
```

So, the fork() system call will always return different kinds of values (As you should know from Question4). One of the values returned would be "0", the other will be a positive non-zero value.
The process that receives the "0" value is the child process.
The process that receives the positive non-zero value is the parent process.
Based on these values, we can then program our code in such a manner that both parent and child can do their own jobs and not appear to be doing the same things.
Why 0? Why Non-Zero?
The reason for this is that the child can always find out who it's parent is via the getppid() function call. However, it is difficult for the parent to know who it's children are. This can only be done if the value of the child's PID is returned to it at the time of the fork().
Look at the following code structure and implement it:

```
// Fork code 2
#include <unistd.h>
#include <???> // For printf()
int main()
{
  int p;
  printf("Original Process, pid = %d\n", getpid() );
  p = fork();
  if (p == 0)
  {
    printf("Child PID = %d, PPID = %d\n", getpid(), getppid() );
  }
  else
  {
    printf("Parent PID = %d, Child ID = %d\n", getpid(), p);
  }
 }
```

What would happen if we don't use the If/Else conditions and immediately write the two printf() statements?
Make sure you understand the structure of the Code, as well as what are the ways of knowing the following:

1. The PID
2. The Parent PID
3. The Child PID

3.2.1.1 Exercise 1

Modify the fork() *call code 2* above and add a system call for sleep() after both the printf() statements. Give a time of 120 seconds to the sleep call. While both the parent and child are sleeping, open a new terminal and check out the outcome of **pstree** command as well as the **ps au** command. Study the output for your parent and child process in both commands, especially noting the **STAT** column.

3.2.1.2 Exercise 2

Model a fork() call in C so that your program can create a total of **EXACTLY** 6 processes (including the parent). (Note: You may check the number of processes created using the method from Exercise 1 (Section-3.2.1.1 by using a sleep of 60 seconds or more and entering **ps au**, in another terminal). Your process hierarchy should be as follows:

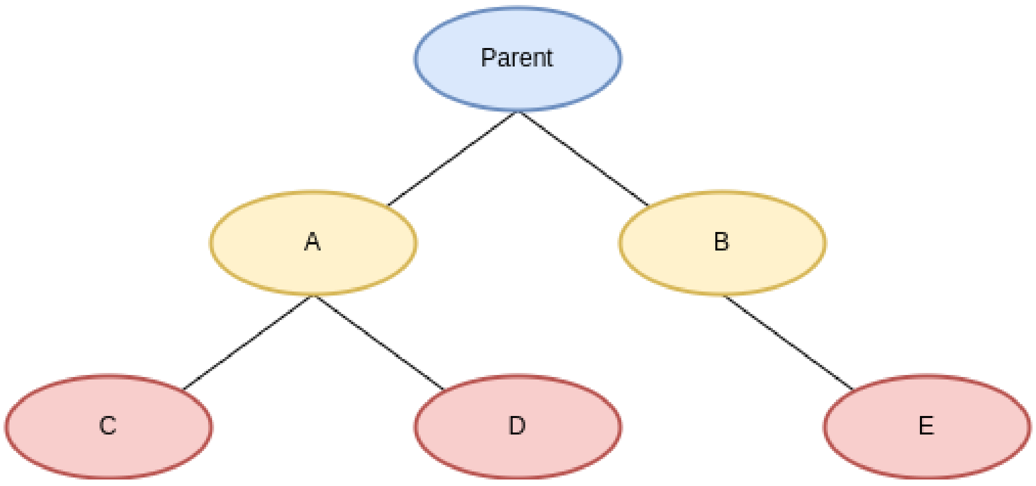


Figure 3.1: Exercise 2 Model

3.2.2 Running States

The job of the exec() call is to ***replace*** the current process with a new process/program. Once the exec() call is made, the current process is "gone" and a new process starts. The Exec() call is actually a family of 6 system calls. Difference between all 6 can be seen from **man 3 exec**

- int execl(const char *path, const char *arg, ...);
- int execlp(const char *file, const char *arg, ...);
- int execle(const char *path, const char *arg, char *const envp[]);
- int execv(const char *path, char *const argv[]);

- int execve(const char *path, const char *argv[], char *const envp[]);
- int execvp(const char *file, char *const argv[]);

Following is usage of one flavor of Exec, the execv() system call:

```
#include <unistd.h>
#include <stdio.h>
int main()
{
  int p;
  char *arg[] = {"/usr/bin/ls", 0};
  p = fork();
  if (p == 0)
  {
    printf("Child Process\n");
    execv(arg[0], arg);
    printf("Child Process\n");
  }
  else
  {
    printf("Parent Process\n");
  }
}
```

Again look at the code. We have referred to Exec() system call but in code we see Execv(). Read man pages for this and find out. To help you understand, try finding answers to the following:

Question1 What is the 1st argument to the execv() call? What is it's contents?

Question2 What is the 2nd argument to it? What is it's contents?

Question3 What is arg?

Question4 Look at the code of the child process (p==0). How many times does the statement "Child Process" appear? Why?

Diagrammatically, the functioning of the exec() system calls would be represented in Figure 3.2, where the dotted line mark the execution and transfer of control whereas the straight lines mark the waiting time.

```
             fork()         wait()
%command ------\_____________/-----> % (Prompt Returns)
                \           /
                 \         /
                  \-------/
             execve()    exit()
```

Figure 3.2: Fork() Exec() representation

3.2.3 Waiting States

3.2.3.1 Sleep()

The sleep() call can be used to cause delay in execution of a program. The delay can be provided as an integer number (representing number of seconds). Usage is simply sleep(int), which will delay the process execution for int seconds. To use sleep(), you will require the *unistd.h* C library.

Exercise

Write a C program that can display a count from 10 to 0 (reverse order) using a for or a while loop. Each number should be displayed after delay of 1 second.

3.2.4 Termination States

A process can terminate in one of the following ways:

1. Main function calls return
2. Exit function called
3. Aborted by higher priority process (e.g., parent or kernel)
4. Terminated by a signal

Regardless of any of these reasons, the same kernel code is invoked which performs the following:

1. Close open files
2. Notifies Parent and Children
3. Release memory resources

3.2.4.1 Exit() Call

The exit() system call causes a normal program to terminate and return status to the parent process. Study the behaviour of the following program. You will see that there may be a number of exit points from a program. Can you identify which exit() call is being used each time a program exits??

```
#include <stdio.h>
#include <stdlib.h>
void anotherExit();
int main()
{
  int num;
  printf("Enter a Number: ");
  scanf("%i", &num);
  if (num>25)
  {
    printf("exit 1\n");
    exit(1);
  }
  else
    anotherExit();
}
void anotherExit()
```

```
{
  printf("Exit 2\n");
  exit(1);
}
```

3.2.4.2 Atexit() Call

The code below provides two functions; atexit() and exit(). Note the structure of code and the behaviour of execution and then attempt the questions in the end.

```
#include<stdio.h>
#include<stdlib.h>
int main(void)
{
  void f1(void), f2(void), f3(void);
  atexit(f1);
  atexit(f2);
  atexit(f3);
  printf("Getting ready to exit\n");
  exit(0);
}
void f1(void)
{
  printf("In f1\n");
}
void f2(void)
{
  printf("In f2\n");
}
void f3(void)
{
  printf("In f3\n");
}
```

Q1 What is the difference between exit() and atexit()? What do they do? (Check *man atexit* and *man 3 exit*).

Q2 What does the 0 provided in the exit() call mean? What will happen if we change it to 1? (Check manual page for exit)

Q3 If we add an exit call to function f1, f2, or f3. What will happen to execution of our program?

Q4 Why do you think we are getting reverse order of execution of atexit calls?

3.2.4.3 Abort Call

Type, run and execute the code below. Then answer the questions in the end.

```
#include <stdio.h>
#include <stdlib.h>
int main()
{
  abort();
  exit(0);
}
```

Q1 Check the man pages for abort. How does the abort call terminate the program? What is the name of the particular signal?

Q2 Execute your program. What is the output of our program?

Q3 Include the abort call in function f3 in our code provided for Atexit() call. How does our program terminate using this?

3.2.4.4 Kill Call

Full description of this call will appear in Section-5.1.2. But as a demonstration, you may run the following code to see how the kill() call works.

```
#include <stdio.h>
#include <sys/types.h>
#include <signal.h>
int main()
{
 printf("Hello");
 kill(getpid(), 9);
 printf("Goodbye");
}
```

You are already familiar with getpid() system call. To find out what 9 is, first look at the output of the command:

```
kill -l
```

Find out the word mentioned next to 9.

3.3 Death of Parent or Child

We covered the process termination using the exit() system call and that a program may have more than one exit points. In a relationship where a process has spawned children, what would be the effect if either the parent dies or the child dies on the other processes linked to it? This section will look and study such a development.

3.3.1 Parent Dies Before Child

If a parent process dies before it's children, the children will be orphaned. They will be assigned to another process in the system and as such the children should be informed about who their new parent is going to be. This new parent process is the parent of all processes in the system, i.e., the "init" process.

```
#include <unistd.h>
#include <stdio.h>
int main()
{
  int i, pid;
  pid = fork();
  if (pid > 0)          // Parent
  {
    sleep(2);
    exit(0);
  }
  else if (pid == 0)    // Child
  {
    for (i=0; i < 5; i++)
```

```
        {
          printf("My parent is %d\n", getppid());
          sleep(1);
        }
      }
    }
```

Run the code.

1. What are the PPID values you are receiving from the for loop?
2. What has happened when the numbers of the PPID change?
3. What is now PID of the init process?

3.3.2 Child Dies Before Parent

Following code is complete opposite of what we have seen so far.

```
#include <unistd.h>
#include <stdio.h>
int main()
{
  int i, pid;
  pid = fork();
  if (pid > 0)         // Parent
  {
    sleep(120);
  }
  else if (pid == 0)   // Child
  {
    exit(0);
  }
}
```

Run the code. In another window (terminal), check the status of your parent and child processes using the "ps au" command. You should be able to see a Z, or <defunct> next to the child process which has been created. Such a process is usually termed a "Zombie" process. Wait for 60 seconds, then check the status of your process again. You will note that both the parent, as well as the Zombie process are now gone.

Chapter 4

Input/Output

Probably the first thing you covered in I/O when you were doing your C/C++ programming courses was **cin**, **cout** or **cerr**. Of these, cout displays something on the standard output (display screen), whereas cin is used for obtaining input from keyboard input device. In linux, there are three types of files that are open all the time for input and output purposes for each process. These are:

1. Standard Input Stream
2. Standard Output Stream
3. Standard Error Stream

Each of these streams are represented by a unique integer number called a *File Descriptor*. In this case, standard input is 0, standard output is 1, and standard error is 2. Other files that are in use by a process will be assigned file descriptor numbers of 3, 4, ... These file descriptors refer to each and every instance of an open file for a process. So, if we want to open a file, close a file, read from a file, or write to a file, it has to be done through it's corresponding file descriptor.

To visualise how this works, we are going to perform a simple exercise. For this, we would require two shells.

- Type the following command and note down the current bash shell PID. Let this be **X**.

  ```
  ps
  ```

- Press CTRL+ALT+T to open a new terminal window.
- The echo command is used to display a line of text.

  ```
  echo "Hello"
  ```

- We are going to use the echo command to write a string *Hello* to a file *hello.txt* using the output redirection method that has been covered in Section-2.4.4. You can view the contents of this file to ensure that the string has indeed been written to it.

  ```
  echo "Hello" > hello.txt
  nano hello.txt
  ```

- In previous bullet, *hello.txt* was a file. We are going to use the same mechanism but a little differently. Replace X with the PID value that you noted down earlier. You will see what /proc directory is used for in later sections. We are specifying that we want to write to file descriptor = 1 of process ID marked by X.

  ```
  echo "Hello" > /proc/X/fd/1
  ```

- Now go back to the previous terminal. Surprise! The string Hello has been written there.

4.1 Open a File

We use the open() call to open a file. Open() can also be used for creating a new file. It's syntax is as such:

```
int open(pathname, flags, modes);
```

This would work by including the *sys/types.h*, *sys/stat.h* and *fcntl.h* C libraries. As parameters, it takes the following:

1. Path name of the file to open
2. Flag specifying how to open it (i.e., for read only, write only, etc.)
3. Access permissions for a file (Provided the file is newly created)

Some of the flags are listed below:

- O_RDONLY for marking a file as read only
- O_WRONLY for marking a file as write only
- O_RDWR for marking a file as read and write
- O_CREAT for creating the new file
- O_EXCL for giving an error when creating a new file and that file already exists

As an example, run the following code for creating a new file:

```
#include <fcntl.h>
#include <stdio.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <unistd.h>

int main(int argc, char* argv[])
{
 char *path = argv[1];
 int fd = open(path, O_WRONLY | O_EXCL | O_CREAT);
 if (fd == -1)
 {
  printf("Error: File not Created\n");
  return 1;
 }
 return 0;
}
```

Compile this, but when running specify the command as such:

```
gcc demo.c
./a.out createThisFile
```

Then run ls and check if the file has been created.

```
ls
```

Give the same command again and check out the output.

Question What is the size of the file? Why is it this size?

4.2 Close a File

So we saw that the open() call returns an integer number (the file descriptor) which has been stored in *fd* variable. Once we are finished with what we are doing with the file, the file should be closed. When a process terminates, linux by default closes all file descriptors so you may not close a file by yourself if you don't want to. But if you do, then read on.

Linux uses a total of 1,024 file descriptors per process by default. So it's a good idea that you close your file descriptor's if they are not used.

To close a file descriptor, just use the following in your code:

```
close(fd);
```

Look at the following code and see what it is the output?

```
#include <fcntl.h>
#include <stdio.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <unistd.h>

int main(int argc, char* argv[])
{
 if (argc != 2)
 {
  printf("Error: Run like this: ");
  printf("%6s name-of-new-file\n", argv[0]);
  return 1;
 }
 char *path = argv[1];
 int i = 0;
 while(i<2)
 {
  int fd = open(path, O_WRONLY | O_CREAT);
  printf("Created! Descriptor is %d\n", fd);
  close(fd);
  i++;
 }
 return 0;
}
```

Comment out the line *close(fd);* and then compile and run again. What is the output this time? Why do you think you are getting different values?

4.3 Writing to a File

Writing to a file is done using the write call. To write, we should obviously open a file first. The syntax of the write() call is as such:

```
write(fd, buffer, size);
```

This would require the *unistd.h* C library. The following are the paremeters used by the call:

- The file descriptor (must be open ... otherwise how are you going to write to a not-open file?)
- Buffer or place where data is located

- Length to write

So let's see the write call in action. Type, compile and run the following code. Then answer the questions at the end.

```
#include <fcntl.h>
#include <stdio.h>
#include <string.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <time.h>
#include <unistd.h>

char* get_timeStamp()
{
 time_t now = time(NULL);
 return asctime(localtime(&now));
}

int main(int argc, char* argv[])
{
 char *filename = argv[1];
 char *timeStamp = get_timeStamp();
 int fd = open(filename, O_WRONLY | O_APPEND | O_CREAT, 0666);
 size_t length = strlen(timeStamp);
 write(fd, timeStamp, length);
 close(fd);
 return 0;
}
```

Q1 What is 0666 that is specified in the open() call? What does it mean?

Q2 What is O_APPEND doing in the same call? Run the program again and check it's output.

Q3 Modify the following line in the code and then compile and run the program and check it's output. What has happened?

```
From:
 size_t length = strlen(timeStamp);
To:
 size_t length = strlen(timeStamp)-5;
```

4.4 Reading from a File

Read() system call is going to be used for this purpose. It's syntax is as such:

```
read(fd, buffer, size);
```

This would require the *unistd.h* C library. the parameters for read() are somewhat the same as that for write(). I.e.,

- The file descriptor (must be open ... otherwise how can you read from a not-open file?)
- Buffer or place where data is to be saved after reading
- Length to read

So let's see the read in action. Type, compile, and run the following code:

```
#include <fcntl.h>
#include <stdio.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <unistd.h>

int main(int argc, char* argv[])
{
 if (argc != 2)
 {
  printf("Error: Run like this: ");
  printf("%6s name-of-existing-file\n", argv[0]);
  return 1;
 }
 char *path = argv[1];
 int fd = open(path, O_RDONLY);
 if (fd == -1)
 {
  printf("File does not exist\n");
  return 1;
 }
 char buffer[200];
 read(fd, buffer, sizeof(buffer)-1);
 printf("Contents of File are:\n");
 printf("%s\n", buffer);
 close(fd);
 return 0;
}
```

Chapter 5

IPC: Signals

Signals are an inter-process communication mechanism provided by operating systems which facilitate communication between processes. It is usually used for notifying a process regarding a certain event. So, we can say that signals are an event-notification system provided by the operating system.

5.1 Signal Delivery Using Kill

Kill is the delivery mechanism for sending a signal to a process. Unlike the name, a kill() call is used only to send a signal to a process. It does not necessarily mean that a process is going to be killed (although it can do exactly that as well). As mentioned earlier, the signal facility is just an event notification facility provided by the operating system. For e.g., a shutdown signal (SIGHUP) can be sent to all processes currently active in the system in order to notify them about a shutdown process event. Upon receipt of this signal, all processes will prepare to terminate. Once the processes terminate, the system can shutdown.

The Kill facility can be used in two ways; as a command from your command prompt, or via the kill() call from your program.

5.1.1 Kill Command

The syntax of the kill command is as such:

```
kill -s PID
```

Here, kill is the command itself, -s is an argument which specifies the type of signal to send, and PID is the integer identifier of the process to which a signal is going to be delivered. The list of signals for -s argument can be seen from the following:

```
kill -l
```

Hence, supposing that we want to terminate a process, we may enter

```
kill -9 12345
```

Where 12345 will be a process id of any active process in the system.

5.1.1.1 Exercise

Let us kill our bash shell using the TERM signal. Find out:

- The integer representation for the SIGTERM signal
- The PID of your current active bash shell using the ps command

Then, use the kill command to send over this signal.

5.1.2 Kill() Call

Usage of the Kill() system call is as such:

```
kill(int, int);
```

For this to work, we will require the *sys/types.h* & *signal.h* C libraries. Here, the 1st parameter is the PID of the process to which signal is to be delivered, and the 2nd parameter is the integer number of signal type (again, can be checkable from *kill -l*). As an example, if we want to send the Terminate signal to the current process, we will use

```
kill(getpid(), SIGKILL)
```

or

```
kill(getpid(), 9)
```

5.1.2.1 Exercise

Write code which does the following:

- Parent process creates a child process using *fork()* call (using proper if/else statement blocks).
- Parent sends the terminate signal via kill() call to child and then waits for 120 seconds

While parent is waiting, the user should check the outcome of ***ps au*** command.

5.1.2.2 Exercise

Modify the code in Exercise-5.1.2.1 such that signal is sent from child to parent. and then have your child wait for 120 seconds.

Again, while child is waiting, the user should check the outcome of ***ps au*** command.

5.2 Signal Handling Using Signal

Signal delivery is handled by the kill command or the kill() call. The process receiving the signal can behave in a number of ways, which is defined by the signal() call.

The syntax of the call is as such:

```
signal(int, conditions)
```

The signal will require the *signal.h* C library to work. From the syntax above, signal() is the system call, the 1st parameter *int* is the integer identifier of the respective signal which is sent, and the last parameter is either of the following:

- **SIG_DFL** which will perform the default mechanism provided by the operating system for that particular signal
- **SIG_IGN** which will ignore that particular signal if it is delivered
- Any function name (for programmer-defined signal handling purposes)

As an example, we are going to send a process the SIGINT signal and count the total number of times that it is received. See the code below for this purpose:

```
#include <signal.h>
#include <stdio.h>
#include <unistd.h>

int sigCounter = 0;

void sigHandler(int sigNum)
{
 printf("Signal received is %d\n", sigNum);
 ++sigCounter;
 printf("Signals received %d\n", sigCounter);
}

int main()
{
 signal(SIGINT, sigHandler);
 while(1)
 {
  printf("Hello Dears\n");
  sleep(1);
 }
 return 0;
}
```

Here, signal is the signal handler call and accepts as input the signal number, and the name of function which is going to behave as signal handler. (If we want it to behave the default way, we specify SIG_DFL, or if we want it to ignore a certain signal, we specify SIG_IGN).

When we run the program, we are instructing our program that if in case the SIGINT signal is detected, we will perform the steps provided in the *sigHandler* function. As long as there is no event, the program will keep on executing the while loop. The event can be delivered by pressing **CTRL+C**.

Try pressing CTRL+C with, and without the signal() call and you will understand the difference yourselves.

Chapter 6

IPC: Pipes

Pipes are another IPC technique for processes to communicate with one another. It can also be used by two threads within the same process to communicate.

In the description of file descriptors from Chapter-4, you will recall that there are three files that are open all the time for input and output purposes. These are:

1. The standard input
2. The standard output
3. The standard error

And that they have a file-descriptor of 0, 1, and 2 respectively. Whenever a program needs to display some output (via cout or printf), it will write that output to the standard output file descriptor. This will in return be displayed on the monitor. Whenever a program needs to take some input from the keyboard, it will take it's input from the standard input file descriptor. Similarly, whenever an error needs to be displayed, that error will be sent to the standard error file descriptor. These files are linked-up internally to peripheral devices such as keyboard, monitor, etc. However, these linkages can be changed to point to something else. Pipes work by doing exactly that! So a pipe will::

- Redirect the standard output of one process to become the standard input of another

The rest of communication is done using the following rules:

- The pipe will be a buffer region in main memory which will be accessible by only two processes.
- One process will read from the buffer while the other will write to it.
- One process cannot read from the buffer unless and untill the other has written to it.

6.1 Pipe On the Shell

Run the following command:

```
pstree
```

As you will notice, the output is too long to fit in the screen. Now run the command with:

```
pstree | less
```

Using the up and down arrows you will notice that you can browse through the output which was otherwise not visible in the first command which we gave. To exit, press **q**.

The | is the symbol for pipe and as you would have guessed, pstree and less are two processes. In this usage, the *standard output of pstree* has become the *standard input of the less* program.

Try the following command:

```
pstree | grep bash
```

Again, you will see that the output of above command is much different from just pstree command. What the above command should print is only those lines of text from the pstree output in which the keyword bash appears. Hence, pstree and grep are two separate processes. But the standard output of the pstree command has become the standard input of the grep program. The pstree command writes out its output to the grep process. The grep process receives it, searches for keyword, formats output, and then displays the result.

6.2 Pipe System Call

Type, compile, and run the following code:

```
#include <unistd.h>
int main()
{
  int pfd[2];
  pipe(pfd);
}
```

The above code creates a pipe using the pipe() system call. We have passed it the name of the integer array *pfd* that we have declared earlier on.

Task: Rewrite this code so that you can view the contents of the array using printf arguments. You should see two numbers. What are these numbers?

Let us extend our code. Type, compile, and run the following:

```
01 #include <unistd.h>
02 int main()
03 {
04   int pid;          // for storing fork() return
05   int pfd[2];       // for pipe file descriptors
06   char aString[20]; // Temporary storage
07   pipe(pfd);        // create our pipe
```

When line number 07 completes, we will have the following in our process:

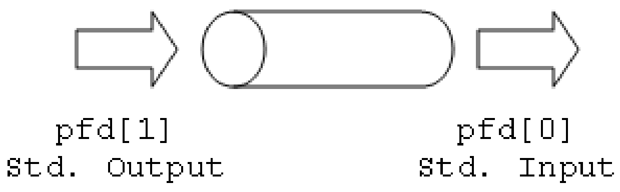
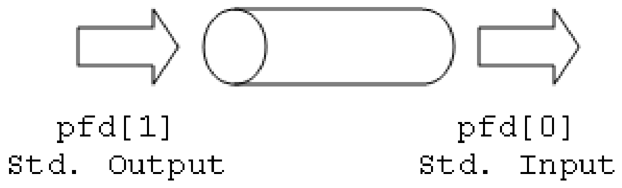


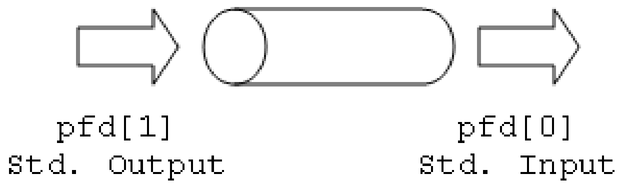
Figure 6.1: Before Fork()

```
08   pid = fork();      // create child process
```

When line number 08 completes, we will have the following in our two processes:



(a) In Child



(b) In Parent

Figure 6.2: After Fork() is made

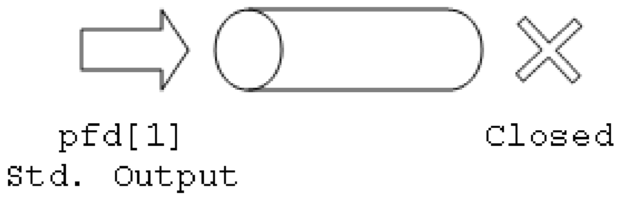
Continuing with rest of code:

```
09   if (pid == 0)                 // For child
10   {
11     write(pfd[1], "Hello", 5);  // Write onto pipe
12   }
13   else                          // For parent
14   {
15     read(pfd[0], aString, 5);   // Read from pipe
16   }
17 }
```

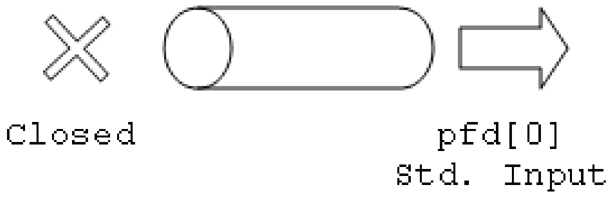
Just like we open a file, we read from a file, we write to a file, and we close a file, we will perform the same operations of open(), close(), read() and write() on the pipe.

Task: Rewrite this code so that you can see the contents of a String in the parent before and after the read() call. What are the contents? You will notice that "Hello" has been mentioned in the chid process. Then how is it possible that we are able to see the term "Hello" in the parent process? The answer is through the pipe mechanism which we just used.

In the code we just saw, since the child is only going to write to a pipe and the parent will only read from the pipe, it makes sense to close the read capabilities for the child and write capabilities for the parent for that particular pipe. Diagramatically, we want to achieve something like the following:



(a) In Child



(b) In Parent

Figure 6.3: Closing un-necessary ends of the Pipe

For that, we have to add the following before line 11:

```
close(pfd[0]);
```

and the following before line 15:

```
close(pfd[1]);
```

6.2.1 Example

Type, run and execute the code below. It should give output which is equivalent to the command ***ls | wc*** (Wc is used for printing three numbers; the number of newlines, the number of words, and the byte count for a file). Note the usage of pipes.

```
#include <unistd.h>
#include <string.h>
#include <stdio.h>
int main()
{
  int pfd[2];
  pipe(pfd);
  if (fork() == 0)
  {
    close(pfd[1]);
    dup2(pfd[0], 0);
    close(pfd[0]);
    execlp("wc", "wc", (char *) 0);
  }
  else
  {
    close(pfd[0]);
    dup2(pfd[1], 1);
    close(pfd[1]);
    execlp("ls", "ls", (char *) 0);
  }
}
```

Part III

Scripting

Chapter 7

Shell Scripting

Shell scripting involves a group of commands grouped together under a single filename. The shell acts as a Command String Interpreter program, interpreting inputs and executing commands interactively or through a script.

7.1 Introduction to Shell Scripting

Shell scripts are dynamically interpreted, not compiled. They allow for automating tasks, executing commands, and manipulating files effectively.

7.1.1 General Elements in Shell Scripting

- **The Shebang Line:** Indicates the shell to be used for interpreting the script.
- **Comments:** Preceded by '#', they are descriptive lines ignored by the shell.
- **Wildcards:** Special characters such as *, ? and [ ] used for filename expansion.

7.2 Bash Operators

7.2.1 Comparison Operators

In Bash, comparison operators are used to compare values. Here are the commonly used operators:

- `-eq`: Equal to
- `-ne`: Not equal to
- `-lt`: Less than
- `-le`: Less than or equal to
- `-gt`: Greater than
- `-ge`: Greater than or equal to

7.2.2 Logical Operators

Logical operators in Bash are used to perform logical operations:

- `-a`: Logical AND
- `-o`: Logical OR
- `!`: Logical NOT

7.2.3 String Comparison Operators

In Bash, string comparison operators are used to compare string values:

- `=`: Equal to
- `!=`: Not equal to
- `<`: Less than
- `>`: Greater than
- `-z`: String is null, that is, has zero length

7.3 Examples

7.3.1 Arithmetic Operations

```bash
#!/bin/bash
echo "enter the a value"
read a
echo "enter the b value"
read b
c=`expr $a + $b`
echo "sum: $c"
c=`expr $a - $b`
echo "sub: $c"
c=`expr $a \* $b`
echo "mul: $c"
c=`expr $a / $b`
echo "div: $c"
```

Listing 7.1: Arithmetic Operations in Shell

7.3.2 Check Even or Odd Numbers

```bash
#!/bin/bash
num="1 2 3 4 5 6 7 8"
for n in $num
do
    q=`expr $n % 2`
    if [ $q -eq 0 ]
    then
        echo "even no"
        continue
    fi
    echo "odd no"
done
```

Listing 7.2: Checking Even or Odd Numbers

7.3.3 Multiplication Table

```
#!/bin/bash
echo "which table you want"
read n
for i in 1 2 3 4 5 6 7 8 9 10
do
    echo "$i * $n = `expr $i \* $n`"
done
```

Listing 7.3: Multiplication Table

7.3.4 Open Files in Loop

```
#!/bin/bash
echo "what do you want"
read n
for i in $(ls)
do
    gedit $i
done
```

Listing 7.4: Open Files in Loop

7.3.5 While Loop Example

```
#!/bin/bash
a=1
while [ $a -lt 11 ]
do
    echo "$a"
    a=`expr $a + 1`
done
```

Listing 7.5: While Loop

7.3.6 If-Else Example

```
#!/bin/bash
for var1 in 1 2 3
do
    for var2 in 0 5
    do
        if [ $var1 -eq 2 -a $var2 -eq 0 ]
        then
            continue
        else
            echo "$var1 $var2"
        fi
    done
done
```

Listing 7.6: If-Else Example

7.3.7 Else-If Example

```
#!/bin/bash
for var1 in 1 2 3
do
    for var2 in 0 5
    do
        if [ $var1 -eq 2 -a $var2 -eq 0 ]
        then
            continue
        else if [ $var1 -eq 4 -a $var2 -eq 1 ]
        then
            echo "$var1"
        else
            echo "$var1 $var2"
        fi
        fi
    done
done
```

Listing 7.7: Else-If Example

7.3.8 Functions

```
#!/bin/bash
add() {
    c=`expr $1 + $2`
    echo "addition = $c"
}
add 5 10
```

Listing 7.8: Functions

7.3.9 Switch Example

```
#!/bin/bash
ch='y'
while [ $ch = 'y' ]
do
    echo "enter your choice"
    echo "1. number of users logged on"
    echo "2. print calendar"
    echo "3. print date"
    read d
    case $d in
        1) who;;
        2) cal 20;;
        3) date;;
        *) break;;
    esac
    echo "do you wish to continue (y/n)"
    read ch
done
```

Listing 7.9: Switch Example

7.4 Exercise

Write a shell script to:

- Create a file with a `.c` extension.
- Copy contents from another file to the new file.
- Provide the user with options to compile, run, or display file contents.

Implement this using both **if** and **switch** statements.

Bibliography

[1] Ubuntu 22.04 handbook.

[2] M. Singhal. *Advanced Concepts in Operating Systems.* Tata-McGraw-Hill, 1994.

[3] W.R. Stevens. *Unix Network Programming: Interprocess Communications*, volume 2. Prentice-Hall India, 2nd edition, 2000.