Learning Report _ Linux Devlopment Tools And Linux System Programing

L&T Technology Services

GLOBAL ENGINEERING ACADEMY

Genesis

Document History

Ver. Rel. No.	Release Date	Prepared. By	Reviewed By	To be approved By	Remarks/Revision Details
1		Name/PS No	Name/PS No	Module Owner Name	Comments
2	26/02/21	99003753			
3					

Introduction :-

On this report we learn Linux devlopment Tools like how to code that interfaces directly with the kernel and core system libraries, including the shell, text editor, compiler, debugger, core utilities, and system daemons. The majority of both Unix and Linux code is still written at the system level, and Linux System Programming focuses on everything above the kernel, where applications such as Apache, bash, cp, vim, Emacs, gcc, gdb, glibc, ls, mv, and X exist.

An overview of Linux, the kernel, the C library, and the C compiler

- Reading from and writing to files, along with other basic file I/O operations, including how the Linux kernel implements and manages file I/O
- Buffer size management, including the Standard I/O library
- Advanced I/O interfaces, memory mappings, and optimization techniques
- The family of system calls for basic process management
- Advanced process management, including real-time processes
- File and directories-creating, moving, copying, deleting, and managing them
- Memory management -- interfaces for allocating memory, managing the memory you have, and optimizing your memory access
- Signals and their role on a Unix system, plus basic and advanced signal interfaces
- Time, sleeping, and clock management, starting with the basics and continuing through POSIX clocks and high resolution timers

Software or Packages Required

- Build essential (GNU tools)
- Valgrind
- gdb
- Make
- git

Tool chain

- Set of Software development tools, linked (or chained)together by specific stages
- Preprocessor, Compiler, Assembler, Linker • Debugger, Symbol Table checker, Object Core dump, header analysis, Size analysis

Native Tool chain

Translates Program for same Hardware

- It is used to make programs for same hardware& OS it is installed on
- It is dependent on Hardware & OS

- It can generate executable file like exe or elf

Cross Tool chain

- Translates Program for different hardware
- It is used to make programs for other hardware like AVR/ARM - It is Independent of Hardware & OS

GNU Compiler Collection includes compilers for C, C++, Objective C, Ada, FORTRAN, Go and java. **gcc :** C compiler in GCC
g++ : C++ compiler in GCC

To check the gcc Version

– gcc -v
– man gcc # More info about gcc

C Program Build Process

1) Pre-processor : gcc -E filename.c cpp hello.c -o hello.i
2) Compilation: gcc -S filename.c gcc -S hello.i
3) Assembler: gcc -c filename.c as -o hello.o hello.s
4) Linker: gcc filename.c ld -o hello.out hello.o ...libraries...

Build Using GCC

Build executable: gcc file.c # Creates a.out as executable file gcc file.c -o output
Creates Output as Executable file

Enable all warning:

gcc -Wall file.c #Enable all Warnings

Enable Debugger Support:

gcc -g file.c #Additional info for debugging purpose

Enable Verbose during compilation:

gcc -v file.c #Info will be printed during compilation

Optimizations on GCC:-

Option	Optimization Level	execution time	code size	memory usage	compile time
-O0	Optimization for compilation time(default)	+	+	-	-
-O1 or -O	Optimization for code size and execution time	-	-	+	+
-O2	Optimization for code size and execution time	--		+	++
-O3	Optimization for code size and execution time	---		+	+++
-Os	Optimization for code size		--		++
-Ofast	O3 with fast non accurate math calculations	---		+	+++

- Utilities -

file: Determines the type of File

```
file hello.c
file hello.o
file hello.out
```

nm: List Symbol Table of Object Files

```
nm hello.o
nm hello.out
```

ldd: List Dynamically Linked Libraries

```
ldd hello.out
```

strip: Remove the symbol table

```
strip hello.out
```

objdump: Information about object files

```
objdump hello.out
```

Libraries :-

Dynamic/Shared Library:

- File extension is **.so** (shared objects) in Unixes or **.dll**(dynamic link library) in Windows.
- Only a small table is created in the executable.
- When the function is called, on demand OS loads machine code of external functions
  - A process known as dynamic linking.
- Executable file is smaller and saves disk space
  - Most operating systems allows one copy of a shared library in memory to be used by all running programs
- Shared library code can be upgraded without the need to recompile your program.

Because of the advantage of dynamic linking, GCC links to the shared library by default if it is available.

Dynamic/Shared Library:

- File extension is **.so** (shared objects) in Unixes or **.dll**(dynamic link library) in Windows.
- Only a small table is created in the executable.
- When the function is called, on demand OS loads machine code of external functions
  - A process known as dynamic linking.
- Executable file is smaller and saves disk space
  - Most operating systems allows one copy of a shared library in memory to be used by all running programs
- Shared library code can be upgraded without the need to recompile your program.

Because of the advantage of dynamic linking, GCC links to the shared library by default if it is available.

Search Path options for GCC (-I, -L and -l) :-

Include path for header files "-I" (Upper case I)

- gcc file.c -Ipath -o Output

Library Path "-L"

- gcc file.c -Lpath -o Output

Library Name "-l" (Lower case L)

- For libmath
  gcc file.c -o Output -lmath

Default Include-paths, Library-paths and Libraries

- cpp -v
- gcc -v hello.c -o hello.out # Lists the Libraries and Paths while compiling

Define Macro "-D"

- Usage : **-Dname=value**
- Value should be enclosed in double quotes if it contains spaces

Environment Variables used by GCC:

PATH:

For searching the executables and run-time shared libraries.

CPATH, C_INCLUDE_PATH or CPLUS_INCLUDE_PATH :

For searching the include-paths for headers.
It is searched after paths specified in -I<dir> options.

LIBRARY_PATH, LD_LIBRARY_PATH:

For searching library-paths for link libraries.
It is searched after paths specified in -L<dir> options.

Variables to select the build tool:

CC, CXX, LD, AS

Flags for the build tools:

CFLAGS, CXXFLAGS, LDFLAGS, ASFLAGS

Static Library Linking :-

Static Library:

```
gcc sum.c -c
gcc sqr.c -c
ar rc libsimple.a sum.o sqr.o

gcc test.c -c
gcc -L. test.o -o s1.out -lsimple
gcc -L. test.o -o s2.out -lsimple -static
```

fun.h

```
#ifndef __FUN_H
#define __FUN_H
int sum(int x, int y);
int square(int x);
#endif
```

test.c

```
#include "fun.h"
#include <stdio.h>
int main() {
  int c, d;
  c = sum(10, 20);
  d = square(10);
  printf("c=%d,d=%d\n",c,d);
  return 0;
}
```

sum.c

```
#include "fun.h"
int sum(int x, int y) {
  return x + y;
}
```

square.c

```
#include "fun.h"
int square(int x) {
return x * x;
}
```

Analysis of Static Library :-

Static Library:

```
file libsimple.a
nm libsimple.a
objdump -d libsimple.a   # -t
readelf -t libsimple.a

file s1.out s2.out
ls -lh s1.out s2.out
size s1.out s2.out
ldd s1.out
ldd s2.out
file s1.out s2.out
objdump -t s2.out
```

```
strip s2.out
objdump -t s2.out
ls -lh s2.out
size s2.out
objdump -t s2.out
strip s2.out
objdump -t s2.out
ls -lh s2.out
size s2.out
```

Dynamic Library Linking :-

Dynamic Library:

```
gcc sum.c -c
gcc sqr.c -c
gcc -shared -o libsample.so sum.o sqr.o

gcc test.c -c
gcc -L. test.o -o d1.out -lsample
LD_LIBRARY_PATH=. ./d1.out
```

fun.h

```
#ifndef __FUN_H
#define __FUN_H
int sum(int x, int y);
int square(int x);
#endif
```

test.c

```
#include "fun.h"
#include <stdio.h>
int main() {
  int c, d;
  c = sum(10, 20);
  d = square(10);
  printf("c=%d,d=%d\n",c,d);
  return 0;
}
```

sum.c

```
#include "fun.h"
int sum(int x, int y) {
  return x + y;
}
```

square.c

```
#include "fun.h"
int square(int x) {
return x * x;
}
```

Usage of Id config to link a dynamic library :

What if we want to install our library so everybody on the system can use it?

- Put the library in a standard location - /usr/lib or /usr/local/lib

```
sudo cp libsample.so /usr/lib
sudo chmod 0755 /usr/lib/libsample.so
sudo ldconfig
unset LD_LIBRARY_PATH
gcc -o test.o d1.out -lsample
```

Versioning of shared object files

```
gcc -shared -Wl,-soname,libsample.so -
o /usr/lib/libample.so.1.0.1 sum.o sqr.o
ln -s /usr/lib/libample.so.1.0.1 /usr/lib/libsample
ln -s -f newfile symlink
   #To update the File that symlink points to
ldconfig -n path
```

Building using Makefile -

test.c

```
#include "fun.h"
#include <stdio.h>
int main() {
  int c, d;
  c = sum(10, 20);
  d = square(10);
  printf("c=%d,d=%d\n",c,d);
  return 0;
}
```

fun.h

```
#ifndef __FUN_H
#define __FUN_H
int sum(int x, int y);
int square(int x);
#endif
```

square.c

```
#include "fun.h"
int square(int x) {
return x * x;
}
```

sum.c

```
#include "fun.h"
int sum(int x, int y)
{
  return x + y;
}
```

Makefile

```
all.out : test.c sum.c sqr.c
    gcc test.c sum.c sqr.c -o all.out

run:all.out
    ./all.out

clean:
    rm all.out
```

Special Variables and User Variables in Makefile -

$@ -Target
$< -First Dependency
$^ -All Dependencies

Makefile

```
TARGET=all.out
OBJS=test.o sum.o sqr.o

all:${TARGET}:

${TARGET}:${OBJS}
    gcc $^ -o $@

test.o:test.c fun.h
    gcc $< -c
sum.o:sum.c fun.h
    gcc $< -c
sqr.o:sqr.c fun.h
    gcc $< -c

clean:
    rm -rf *.o all.out
```

Rules in Makefile -

Pattern Based Rule:

```
%.o:%.c
    gcc $^ -o $@
```

Make already knows how to generate a .o file

Implicit rule

```
%.o:%.c
```

Makefile

```
TARGET=all.out
OBJS=test.o sum.o sqr.o

all:${TARGET}:

${TARGET}:${OBJS}
    gcc $^ -o $@

%.o:%.c fun.h
    gcc $< -c

clean:
    rm -rf *.o all.out
```

GDB:-

Internal commands in gdb shell

- r - run
- c - continue
- q - quit
- s - step (step in)
- n - next (step over)
- f - run up to finish of function (step out)
- b - break point
- Info break – lists the break points
- display x – display value of x after every step or pause.
- bt - back trace of current function
- up
- down
- return – return from current function

Static and Dynamic Code Analysis :-

Static Analysis Tool: **cppCheck**
Usage:

```
cppcheck path_to_src
```

Dynamic Code Analysis tool: **Valgrind**
Usage:

```
valgrind ./a.out
```

Linux System Programing Learning Report

Introduction :-

Unix systems historically did not include many higher-level abstractions. Even programming in a development environment such as the X Window System exposed in full view the core Unix system API. Consequently, it can be said that this book is a book on Linux programming in general. But note that this book does not cover the Linux programming *environment* —for example, there is no tutorial on *make* in these pages. What is covered is the system programming API exposed on a modern Linux machine.

We can compare and contrast system programming with application programming, which differ in some important aspects but are quite similar in others. System programming's hallmark is that the system programmer must have an acute awareness of the hardware and the operating system on which they work. Where system programs interface primarily with the kernel and system libraries, application programs also interface with high-level libraries. These libraries *abstract* away the details of the hardware and operating system. Such abstraction has several goals: portability with different systems, compatibility with different versions of those systems, and the construction of higher-level toolkits that are easier to use, more powerful, or both. How much of a given application uses system versus high-level libraries depends on the level of the stack at which the application was written. Some applications are written exclusively to higher-level abstractions. But even these applications, far from the lowest levels of the system, benefit from a programmer with knowledge of system programming. The same good practices and understanding of the underlying system inform and benefit all forms of programming.

Cornerstones of System Programming :-

There are three cornerstones of system programming in Linux: system calls, the C library, and the C compiler. Each deserves an introduction.

System Calls

System programming starts and ends with ***system calls***. System calls (often shortened to ***syscalls***) are function invocations made from user space—your text editor, favorite game, and so on—into the kernel (the core internals of the system) in order to request some service or resource from the operating system. System calls range from the familiar, such as read() and write(), to the exotic, such as get_thread_area() and set_tid_address().

Linux implements far fewer system calls than most other operating system kernels. For example, a count of the x86-64 architecture's system calls comes in at around 300, compared with the suspected thousands of system calls on Microsoft Windows. In the Linux kernel, each machine architecture (such as Alpha, x86-64, or PowerPC) can augment the standard system calls with its own. Consequently, the system calls available on one architecture may differ from those available on another. Nonetheless, a very large subset of system calls—more than 90 percent—is implemented by all architectures. It is this shared subset, these common interfaces, that we cover in this book.

Invoking system calls

It is not possible to directly link user-space applications with kernel space. For reasons of security and reliability, user-space applications must not be allowed to directly execute kernel code or manipulate kernel data. Instead, the kernel must provide a mechanism by which a user-space application can "signal" the kernel that it wishes to invoke a system call. The application can then ***trap*** into the kernel through this well-defined mechanism and execute only code that the kernel allows it to execute. The exact mechanism varies from architecture to architecture. On i386, for example, a user-space application executes a software interrupt instruction, int, with a value of 0x80. This instruction causes a switch into kernel space, the protected realm of the kernel, where the kernel executes a software interrupt handler—and what is the handler for interrupt 0x80? None other than the system call handler!

The application tells the kernel which system call to execute and with what parameters via ***machine registers***. System calls are denoted by number, starting at 0. On the i386 architecture, to request system call 5 (which happens to be open()), the user-space application stuffs 5 in register eax before issuing the int instruction.

Parameter passing is handled in a similar manner. On i386, for example, a register is used for each possible parameter—registers ebx, ecx, edx, esi, and edi contain, in order, the first five parameters. In the rare event of a system call with more than five parameters, a single register is used to point to a buffer in user space where all of the parameters are kept. Of course, most system calls have only a couple of parameters.

Other architectures handle system call invocation differently, although the spirit is the same. As a system programmer, you usually do not need any knowledge of how the kernel handles system call invocation. That knowledge is encoded into the standard calling conventions for the architecture, and handled automatically by the compiler and the C library.

The C Library

The C library (***libc***) is at the heart of Unix applications. Even when you're programming in another language, the C library is most likely in play, wrapped by the higher-level libraries, providing core services, and facilitating system call invocation. On modern Linux systems, the C library is provided by ***GNU libc***, abbreviated ***glibc***, and pronounced ***gee-lib-see*** or, less commonly, ***glib-see***.

The GNU C library provides more than its name suggests. In addition to implementing the standard C library, ***glibc*** provides wrappers for system calls, threading support, and basic application facilities.

The C Compiler

In Linux, the standard C compiler is provided by the ***GNU Compiler Collection*** (***gcc***). Originally, ***gcc*** was GNU's version of ***cc***, the ***C Compiler***. Thus, ***gcc*** stood for ***GNU C Compiler***. Over time, support was added for more and more languages. Consequently, nowadays ***gcc*** is used as the generic name for the family of GNU compilers. However, ***gcc*** is also the binary used to invoke the C compiler. In this book, when I talk of ***gcc***, I typically mean the program ***gcc***, unless context suggests otherwise.

The compiler used in a Unix system—Linux included—is highly relevant to system programming, as the compiler helps implement the C standard (see C Language Standards) and the system ABI (see APIs and ABIs).

C++

This chapter focuses on C as the lingua franca of system programming, but C++ plays a significant role.

To date, C++ has taken a backseat to C in system programming. Historically, Linux developers favored C over C++: core libraries, daemons, utilities, and of course the Linux kernel are all written in C. Where the ascendancy of C++ as a "better C" is all but universal in most non-Linux environments, in Linux C++ plays second fiddle to C.

Nonetheless, in much of this book, you can replace "C" with "C++" without issue. Indeed, C++ is an excellent alternative to C, suitable for any system programming task: C++ code can link to C code, invoke Linux system calls, and utilize ***glibc***.

C++ programming adds two more cornerstones to the system programming foundation: the standard C++ library and the GNU C++ compiler. The ***standard C++ library*** implements C++ system interfaces and the ISO C++11 standard. It is provided by the ***libstdc++*** library (sometimes written ***libstdcxx***). The ***GNU C++ compiler*** is the standard compiler for C++ code on Linux systems. It is provided by the ***g++*** binary.

APIs and ABIs

Programmers are naturally interested in ensuring their programs run on all of the systems that they have promised to support, now and in the future. They want to feel secure that programs they write on their Linux distributions will run on other Linux distributions, as well as on other supported Linux architectures and newer (or earlier) Linux versions.

At the system level, there are two separate sets of definitions and descriptions that impact portability. One is the ***application programming interface*** (API), and the other is the ***application binary interface*** (ABI). Both define and describe the interfaces between different pieces of computer software.

APIs

An API defines the interfaces by which one piece of software communicates with another at the source level. It provides abstraction by providing a standard set of interfaces——usually functions—that one piece of software (typically, although not necessarily, a higher-level piece) can invoke from another piece of software (usually a lower-level piece). For example, an API might abstract the concept of drawing text on the screen through a family

of functions that provide everything needed to draw the text. The API merely defines the interface; the piece of software that actually provides the API is known as the ***implementation*** of the API.

It is common to call an API a "contract." This is not correct, at least in the legal sense of the term, as an API is not a two-way agreement. The API user (generally, the higher-level software) has zero input into the API and its implementation. It may use the API as-is, or not use it at all: take it or leave it! The API acts only to ensure that if both pieces of software follow the API, they are ***source compatible***; that is, that the user of the API will successfully compile against the implementation of the API.

A real-world example of an API is the interfaces defined by the C standard and implemented by the standard C library. This API defines a family of basic and essential functions, such as memory management and string manipulation routines.

Throughout this book, we will rely on the existence of various APIs, such as the standard I/O library discussed in Chapter 3. The most important APIs in Linux system programming are discussed in the section Standards.

ABIs

Whereas an API defines a source interface, an ABI defines the binary interface between two or more pieces of software on a particular architecture. It defines how an application interacts with itself, how an application interacts with the kernel, and how an application interacts with libraries. Whereas an API ensures source compatibility, an ABI ensures ***binary compatibility***, guaranteeing that a piece of object code will function on any system with the same ABI, without requiring recompilation.

ABIs are concerned with issues such as calling conventions, byte ordering, register use, system call invocation, linking, library behavior, and the binary object format. The calling convention, for example, defines how functions are invoked, how arguments are passed to functions, which registers are preserved and which are mangled, and how the caller retrieves the return value.

Although several attempts have been made at defining a single ABI for a given architecture across multiple operating systems (particularly for i386 on Unix systems), the efforts have not met with much success. Instead, operating systems—Linux included—tend to define their own ABIs however they see fit. The ABI is intimately

tied to the architecture; the vast majority of an ABI speaks of machine-specific concepts, such as particular registers or assembly instructions. Thus, each machine architecture has its own ABI on Linux. In fact, we tend to call a particular ABI by its machine name, such as ***Alpha***, or ***x86-64***. Thus, the ABI is a function of both the operating system (say, Linux) and the architecture (say, x86-64).

System programmers ought to be aware of the ABI but usually need not memorize it. The ABI is enforced by the ***toolchain***—the compiler, the linker, and so on—and does not typically otherwise surface. Knowledge of the ABI, however, can lead to more optimal programming and is required if writing assembly code or developing the toolchain itself (which is, after all, system programming).

The ABI is defined and implemented by the kernel and the toolchain.

Standards

Unix system programming is an old art. The basics of Unix programming have existed untouched for decades. Unix systems, however, are dynamic beasts. Behavior changes and features are added. To help bring order to chaos, standards groups codify system interfaces into official standards. Numerous such standards exist but, technically speaking, Linux does not officially comply with any of them. Instead, Linux ***aims*** toward compliance with two of the most important and prevalent standards: POSIX and the Single UNIX Specification (SUS).

POSIX and SUS document, among other things, the C API for a Unix-like operating system interface. Effectively, they define system programming, or at least a common subset thereof, for compliant Unix systems.

POSIX and SUS History

In the mid-1980s, the Institute of Electrical and Electronics Engineers (IEEE) spearheaded an effort to standardize system-level interfaces on Unix systems. Richard Stallman, founder of the Free Software movement, suggested the standard be named ***POSIX*** (pronounced ***pahz-icks***), which now stands for ***Portable Operating System Interface***.

The first result of this effort, issued in 1988, was IEEE Std 1003.1-1988 (POSIX 1988, for short). In 1990, the IEEE revised the POSIX standard with IEEE Std 1003.1-1990 (POSIX 1990). Optional real-time and threading support were documented in, respectively, IEEE Std 1003.1b-1993 (POSIX 1993 or POSIX.1b), and IEEE Std 1003.1c-1995 (POSIX 1995 or POSIX.1c). In 2001, the optional standards were rolled together with the base

POSIX 1990, creating a single standard: IEEE Std 1003.1-2001 (POSIX 2001). The latest revision, released in December 2008, is IEEE Std 1003.1-2008 (POSIX 2008). All of the core POSIX standards are abbreviated POSIX.1, with the 2008 revision being the latest.

In the late 1980s and early 1990s, Unix system vendors were engaged in the "Unix Wars," with each struggling to define its Unix variant as ***the*** Unix operating system. Several major Unix vendors rallied around The Open Group, an industry consortium formed from the merging of the Open Software Foundation (OSF) and X/Open. The Open Group provides certification, white papers, and compliance testing. In the early 1990s, with the Unix Wars raging, The Open Group released the Single UNIX Specification (SUS). SUS rapidly grew in popularity, in large part due to its cost (free) versus the high cost of the POSIX standard. Today, SUS incorporates the latest POSIX standard.

The first SUS was published in 1994. This was followed by revisions in 1997 (SUSv2) and 2002 (SUSv3). The latest SUS, SUSv4, was published in 2008. SUSv4 revises and combines IEEE Std 1003.1-2008 and several other standards. Throughout this book, I will mention when system calls and other interfaces are standardized by POSIX. I mention POSIX and not SUS because the latter subsumes the former.

C Language Standards

Dennis Ritchie and Brian Kernighan's famed book, ***The C Programming Language*** (Prentice Hall), acted as the informal C specification for many years following its 1978 publication. This version of C came to be known as ***K&R C***. C was already rapidly replacing BASIC and other languages as the lingua franca of microcomputer programming. Therefore, to standardize the by-then quite popular language, in 1983 the American National Standards Institute (ANSI) formed a committee to develop an official version of C, incorporating features and improvements from various vendors and the new C++ language. The process was long and laborious, but ***ANSI C*** was completed in 1989. In 1990, the International Organization for Standardization (ISO) ratified ***ISO C90***, based on ANSI C with a small handful of modifications.

In 1995, the ISO released an updated (although rarely implemented) version of the C language, ***ISO C95***. This was followed in 1999 with a large update to the language, ***ISO C99***, that introduced many new features, including inline functions, new data types, variable-length arrays, C++-style comments, and new library functions. The latest version

of the standard is ***ISO C11***, the most significant feature of which is a formalized memory model, enabling the portable use of threads across platforms.

On the C++ front, ISO standardization was slow in arriving. After years of development—and forward incompatible compiler release—the first C standard, ISO C98, was ratified in 1998. While it greatly improved compatibility across compilers, several aspects of the standard limited consistency and portability. ***ISO C++03*** arrived in 2003. It offered bug fixes to aid compiler developers but no user-visible changes. The next and most recent ISO standard, ***C++11*** (formerly ***C++0x*** in suggestion of a more optimistic release date), heralded numerous language and standard library additions and improvements—so many, in fact, that many commentators suggest C++11 is a distinct language from previous C++ revisions.

Linux and the Standards

As stated earlier, Linux aims toward POSIX and SUS compliance. It provides the interfaces documented in SUSv4 and POSIX 2008, including real-time (POSIX.1b) and threading (POSIX.1c) support. More importantly, Linux strives to behave in accordance with POSIX and SUS requirements. In general, failing to agree with the standards is considered a bug. Linux is believed to comply with POSIX.1 and SUSv3, but as no official POSIX or SUS certification has been performed (particularly on each and every revision of Linux), we cannot say that Linux is officially POSIX- or SUS-compliant.

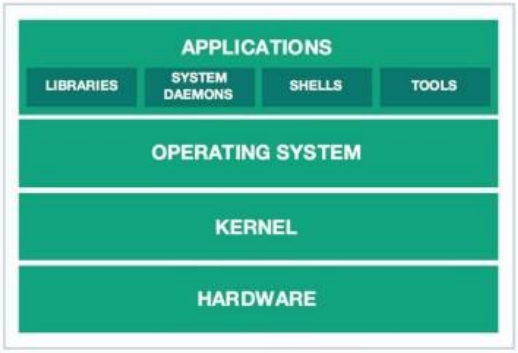
With respect to language standards, Linux fares well. The ***gcc*** C compiler is ISO C99-compliant; support for C11 is ongoing. The ***g++*** C++ compiler is ISO C++03-compliant with support for C++11 in development. In addition, ***gcc*** and ***g++***_ implement extensions to the C and C++ languages. These extensions are collectively called ***GNU C***, and are documented in Appendix A.

Linux has not had a great history of forward compatibility,[1] although these days it fares much better. Interfaces documented by standards, such as the standard C library, will obviously always remain source compatible. Binary compatibility is maintained across a given major version of ***glibc***, at the very least. And as C is standardized, ***gcc*** will always compile legal C correctly, although ***gcc***-specific extensions may be deprecated and eventually removed with new ***gcc*** releases. Most importantly, the Linux kernel guarantees the stability of system calls. Once a system call is implemented in a stable version of the Linux kernel, it is set in stone.

Among the various Linux distributions, the Linux Standard Base (LSB) standardizes much of the Linux system. The LSB is a joint project of several Linux vendors under the auspices of the Linux Foundation (formerly the Free Standards Group). The LSB extends POSIX and SUS, and adds several standards of its own; it attempts to provide a binary standard, allowing object code to run unmodified on compliant systems. Most Linux vendors comply with the LSB to some degree.

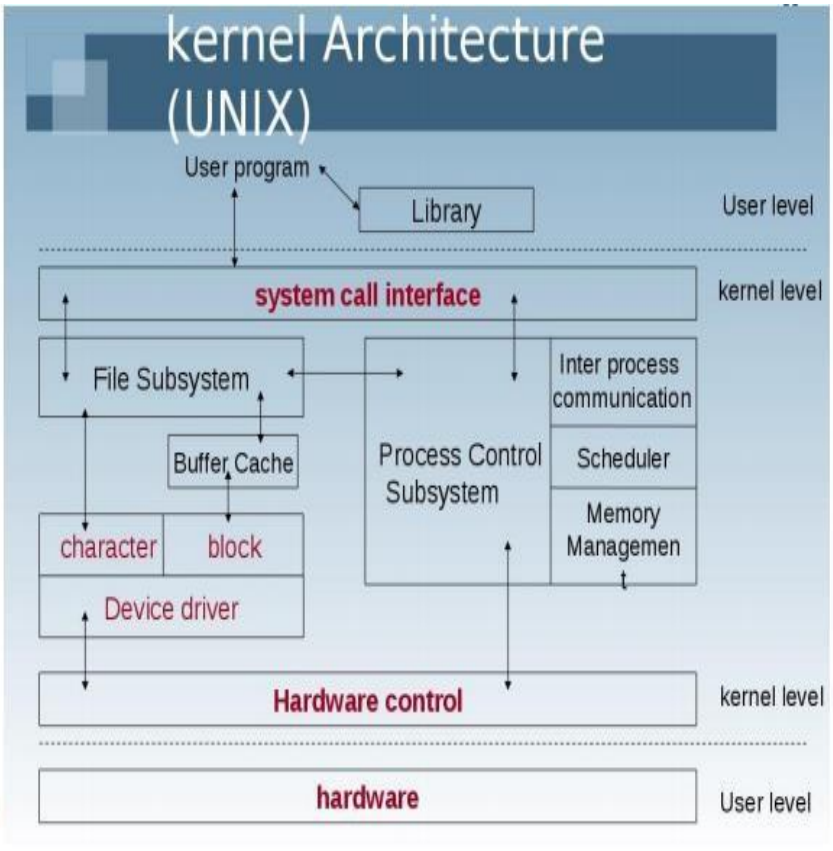
Linux OS Architecture : -

- Kernel
  - Core of OS, responsible for all major OS activities, interacts with hardware, provides abstraction to hardware from system / application programs
- Drivers
  - Used for interaction for additional hardware & I/O
- System Libraries
  - Special programs used by system / application programs access kernel's features, implement most of the functionalities of OS
  - Multimedia library, Network library
- System Utilities
  - Used to do specialized, individual level tasks
  - Shell, Terminal

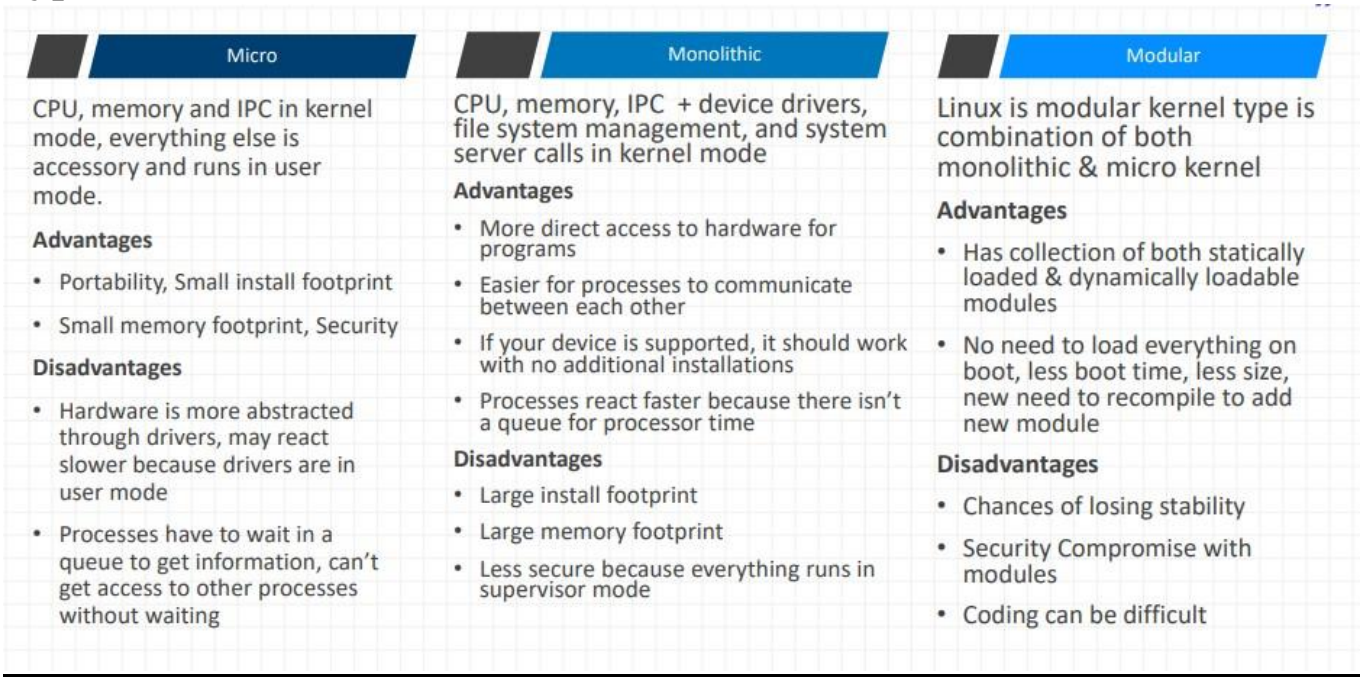


Kernal -

- Mandate component of Operating System
- Resides in memory all the time, rest all depending on kernel
- Provides basic services including memory management, IO management & other management services
- Provides services to application and libraries in the form of SYSTEM CALLS
- Detailed Info: https://en.wikipedia.org/wiki/Linux_kernel



Types of Kernel:-

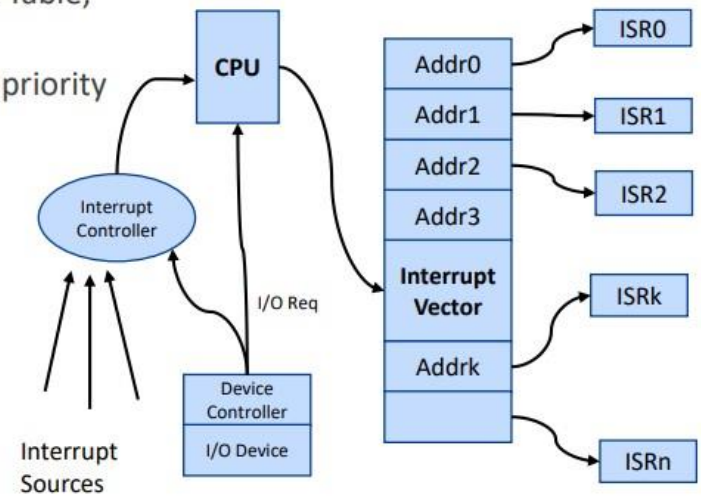


Linux OS Kernel :-

- Compressed Kernel is stored at /boot/vmlinu*
- Dynamic modules of kernel /lib/modules
- uname –r
  - 5.4.0-33-generic (major.minor.release-tagname)
- Versions of kernel
  - 2.x, 2.4, 2.5, 2.6, 3.x, 4.x, 5.x
  - 5.x is current version

Interrupts :-

- Asynchronous events
- IRQ (Interrupt Request), Interrupt Vector Table, ISR (Interrupt Service Routine)
- Interrupts must be serviced with utmost priority
- ISR should be as short as possible with no/minimal blocking calls
- Maskable & Non-maskable Interrupts
- Types
  - Hardware Interrupts
  - Software Interrupts

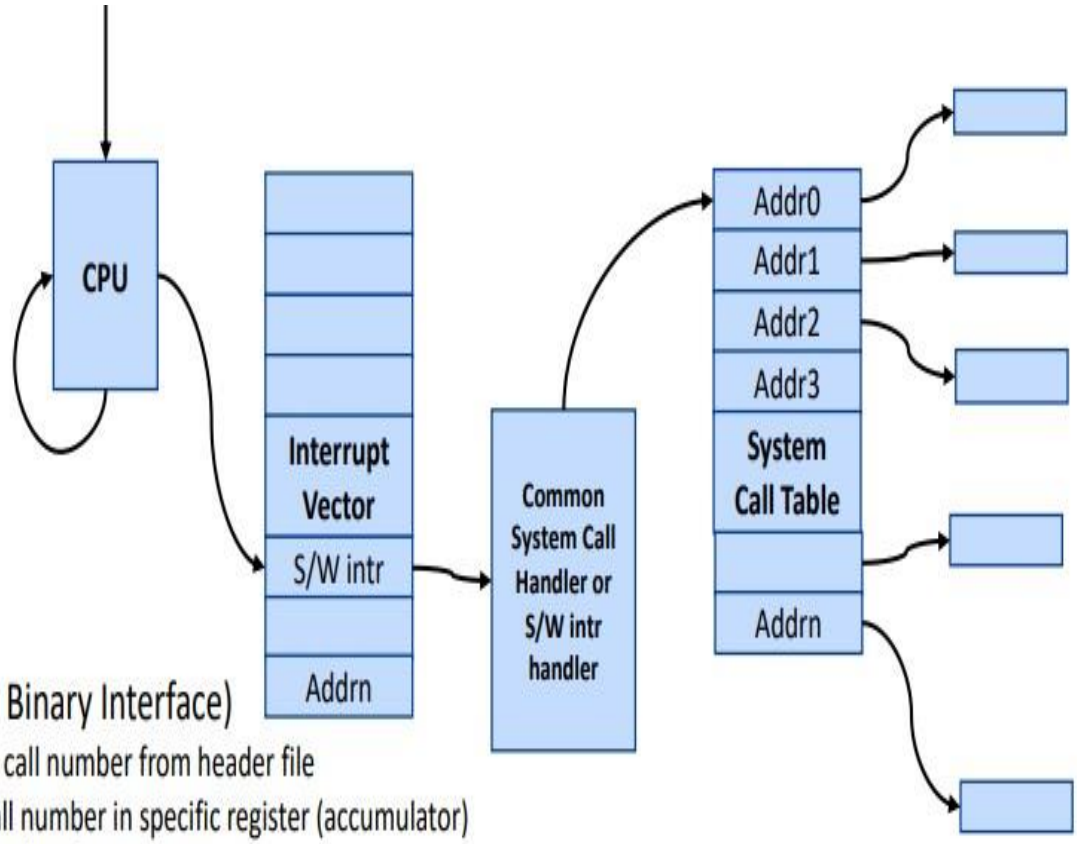


- Interface to OS Services, Communication between Kernel mode and User mode.
- System calls initiated by user space, executed by kernel space
- System calls are also referred as software interrupts
- Identified by Unique Number
- Written in C or Assemble within Kernel space
- System call offers the services of the operating system to the user programs via API (Application Programming Interface)
- Follows “standard protocol” for parameter flow and return values
  - No common memory between user space and kernel space, hence system calls use REGISTERS for communications
  - If arguments are more than available registers, then arguments are packed in structures or blocks and address is passed in register
- man syscall → details of registers

System calls :-

- Interface to OS Services, Communication between Kernel mode and User mode.
- System calls initiated by user space, executed by kernel space
- System calls are also referred as software interrupts
- Identified by Unique Number
- Written in C or Assemble within Kernel space
- System call offers the services of the operating system to the user programs via API (Application Programming Interface)
- Follows "standard protocol" for parameter flow and return values
  - No common memory between user space and kernel space, hence system calls use REGISTERS for communications
  - If arguments are more than available registers, then arguments are packed in structures or blocks and address is passed in register
- man syscall → details of registers

Application Binary Interface –

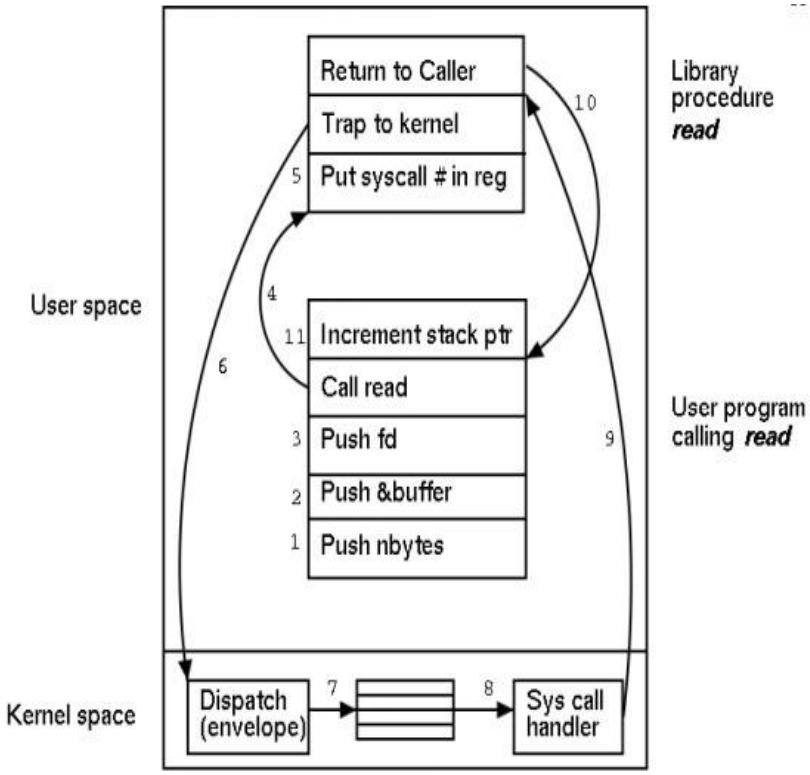


- ABI (Application Binary Interface)
  - Identify system call number from header file
  - Store system call number in specific register (accumulator)
  - Store parameters in other registers
  - Initiate TRAP instruction
- On execution, system call always returns 0 or positive for SUCCESS & negative for FAIL

Types of System calls –

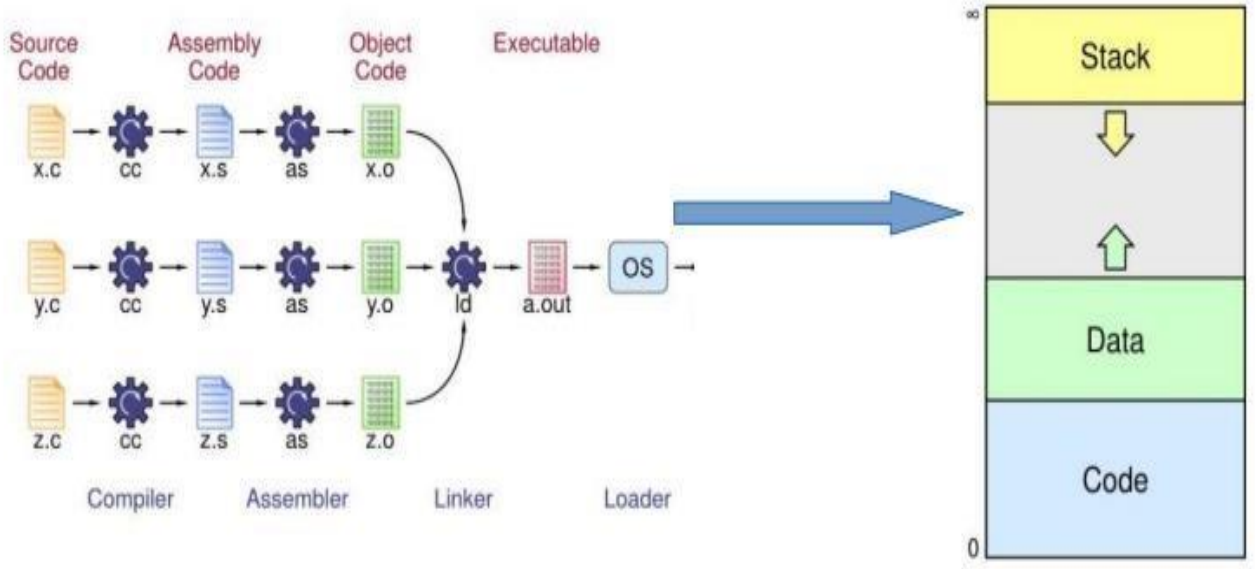
Types of System calls
- Process Control
- File Management
- Device Management
- Information Maintenance
- Communications
- Protection

- Write system call:
  - printf in C, echo in shell, cout in C++
- Read system call:
  - scanf in C, cin in C++
- Trace the system calls:
  - strace man, echo, cp, cat
  - man strace
- list of system call numbers
  - /usr/include/asm*/unistd.h

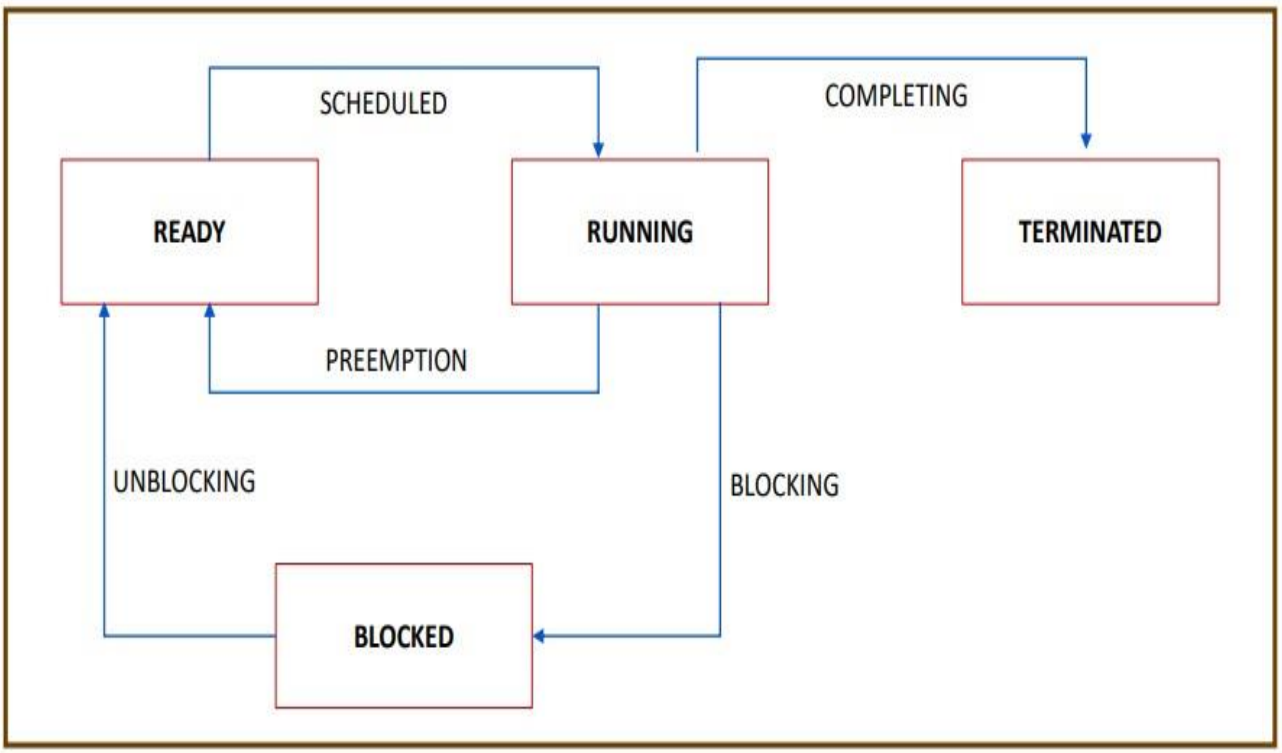


Processs Management

- Process is a Program under execution
- Program is passive entity and process is active entity
- Every process has its own independent stack
- Kernel maintains process list table in the form of doubly linked list
- Each process has a unique id (pid)

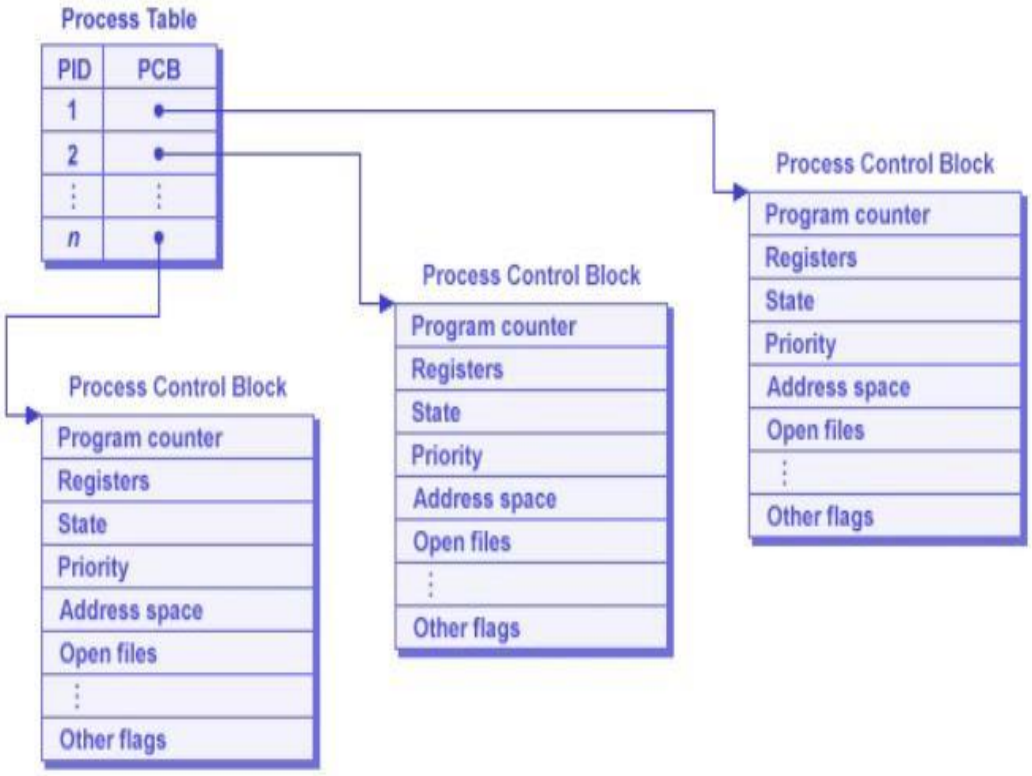


Process Life Cycle -



Process table and process Control Block -

- Process table is maintained by kernel
- Process Control Block



Process :-

- To start a process in Background use **&** symbol in command
  - cat file.txt &
  - jobs
- Lis the running processes
  - ps , ps –f
- Stop a process
  - kill -9, pkill
- Parent and Child
  - Each Process(PID) has a parent (PPID)
- Zombie and Orphan Process
  - Orphan process is one whose parent is killed/terminated before itself.
  - Processes which completed the execution but still have entry in process table.
- Daemon Processes
  - Processes that run in background

Process Hierachy :-

- Every process has parent process
- a.out → shell → terminal → .. → init with PID = 1
- init is considered as the ORIGIN of linux process hierarchy
- Commonly used commands
  - pstree, pstree –np
  - top
  - ps, ps –el, ps aux, ps –e –o pid,ppid,stat,cmd
  - pgrep
  - kill, killall, kill -9, pkill

New Process Creation -

fork

- Creates a new process known as child process
- New pid, process control block (PCB) / process descriptor (PD) will be allocated to child (new entry in process table)
- Duplicates resources from parent to child
- fork returns zero to child, non zero to parent
- Child resumes from next statement after fork
- Parent & child run concurrently based on architecture

Process Termination

- exit() function causes normal process termination and the value of status is returned to the parent
- Process normal termination can be
  - success – exit (0)
  - Failure – exit with positive value
- abnormal termination
  - With exceptions

Waitpid -

- Blocks parent process till completion of child process
- Collect exit status of child
- Cleans some pending resources of child (else child will become Zombie)
- waitpid paramaters
  - 1st param : pid of child process waiting for, -1 means any one child
  - 2nd param : status of terminated child (pass by address)
  - 3rd param : flags
- man waitpid