./

Learning Report – Linux Development Tools

And Linux System Programing



|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **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 | 27/02/21 | 99003751 |  |  |  |
| 3 |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |

**Document History**

**Introduction:-**

On this report we learn Linux development 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 programsfor other hardware like AVR/ARM
* It is Independentof 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

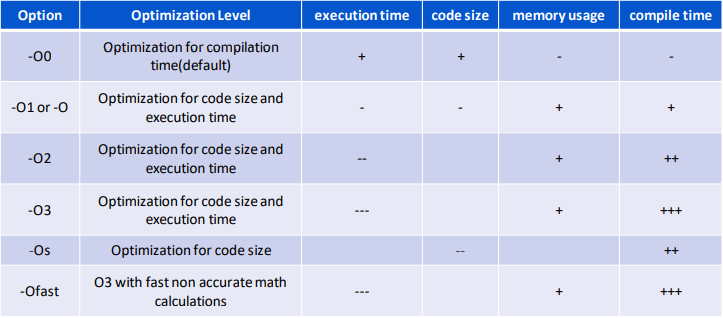
**Enable Debugger Support:**

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

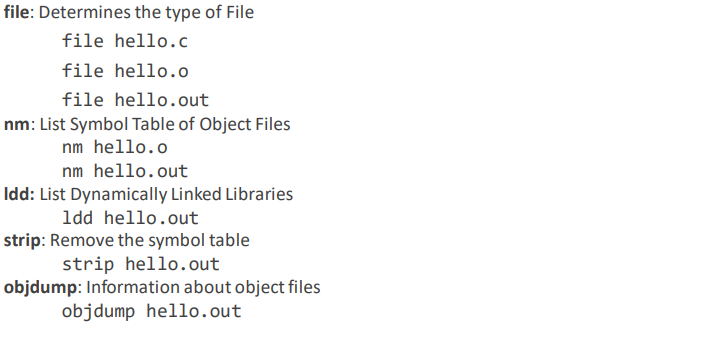
**Enable Verbose during compilation:**

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

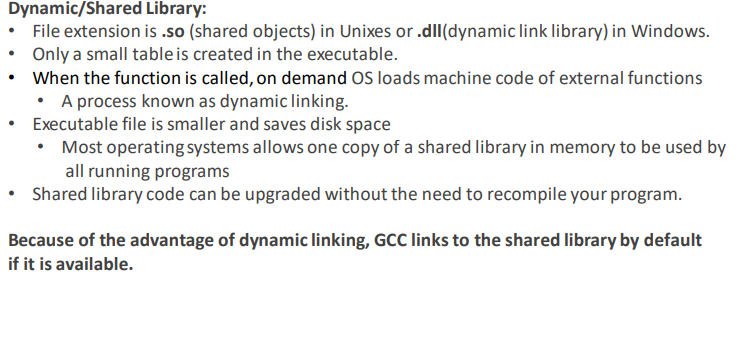
**Optimzations on gcc:-**

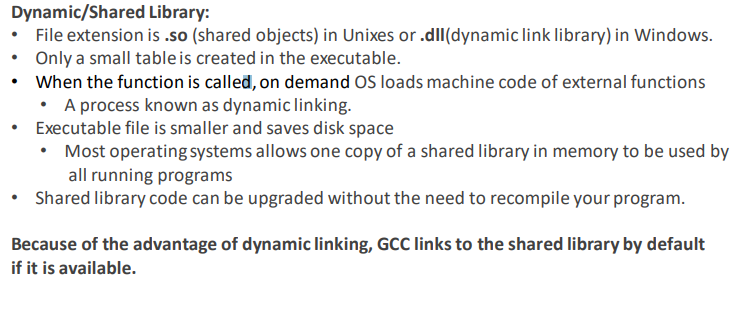
****

**Utilities:-**

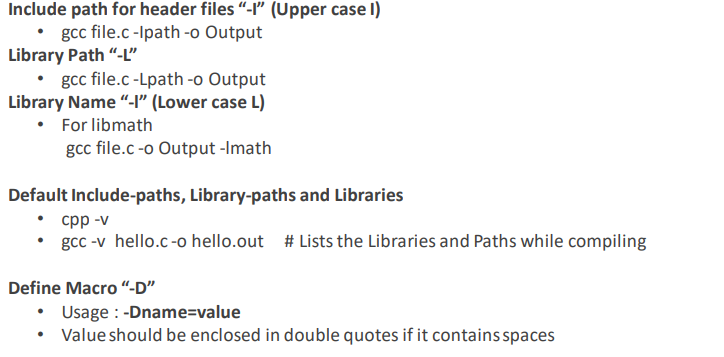
****

**Libraries:-**

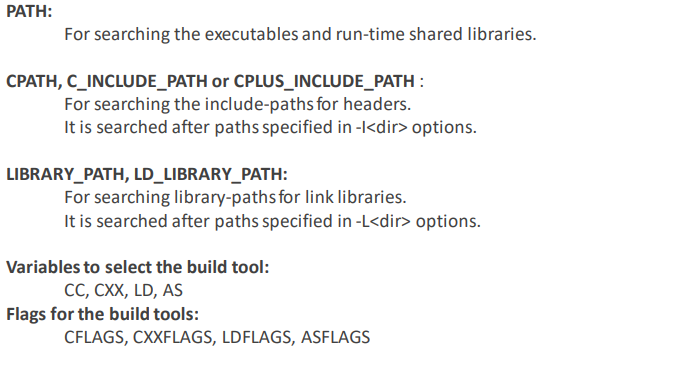
****



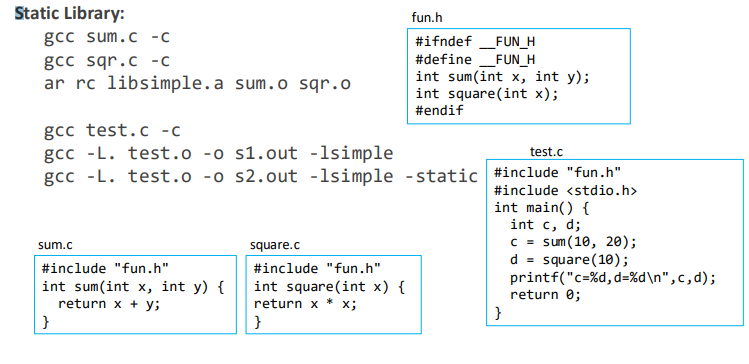
**Search Path options for GCC (-I, -L and -l):-**



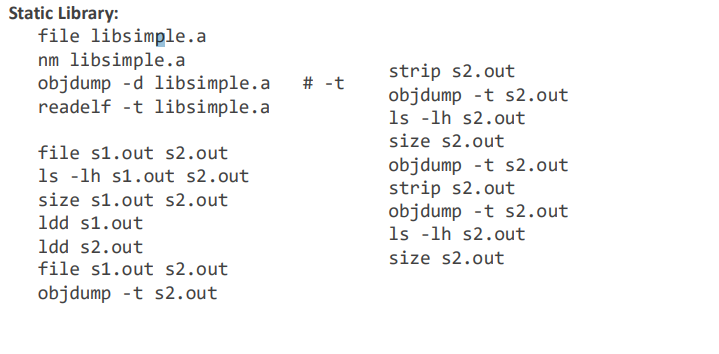
**Environment Variables used by GCC:-**

****

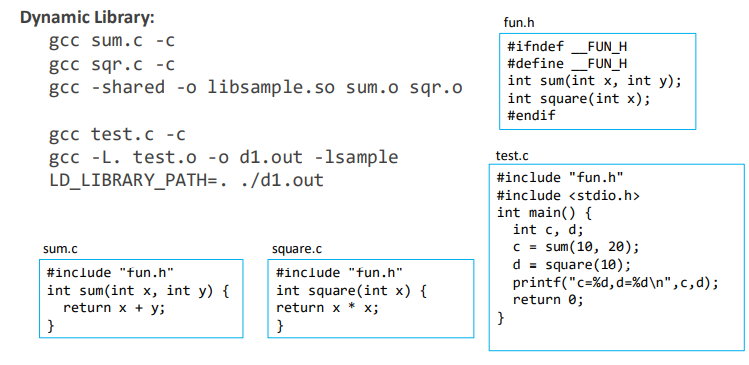
**Static Library Linking:-**

****

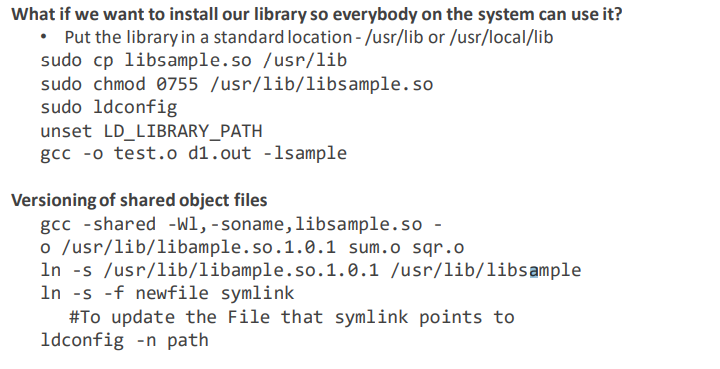
**Analysis of Static Library:-**



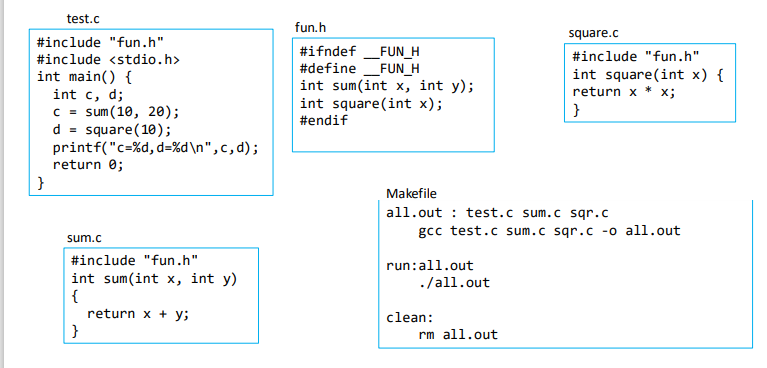
**Dynamic Library Linking:-**



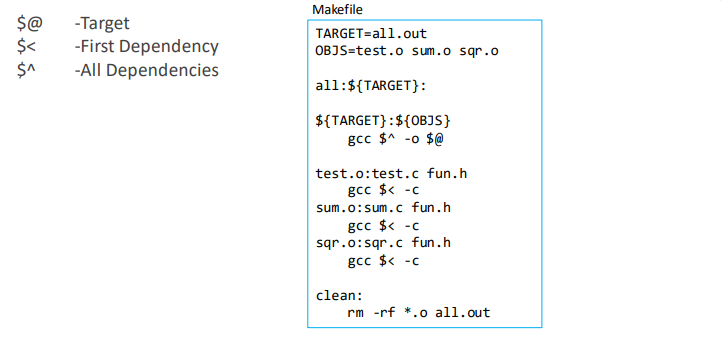
**Usage of ldconfig to link a dynamic library:-**

****

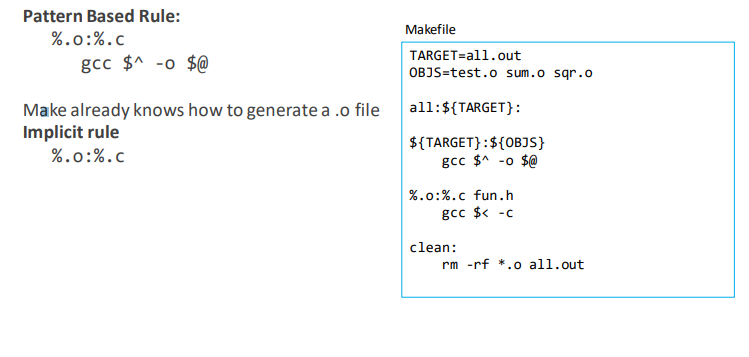
# Building using Makefile:-



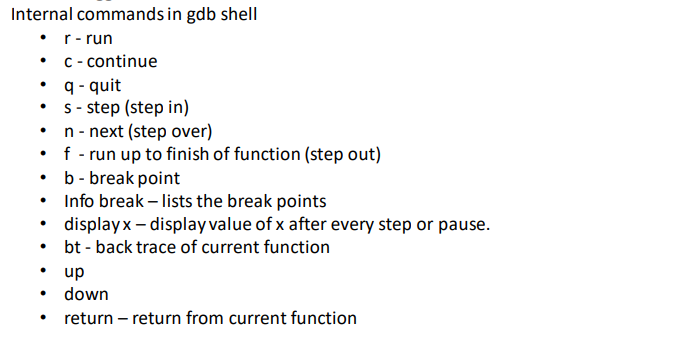
# Special Variables and User Variables in Makefile:-



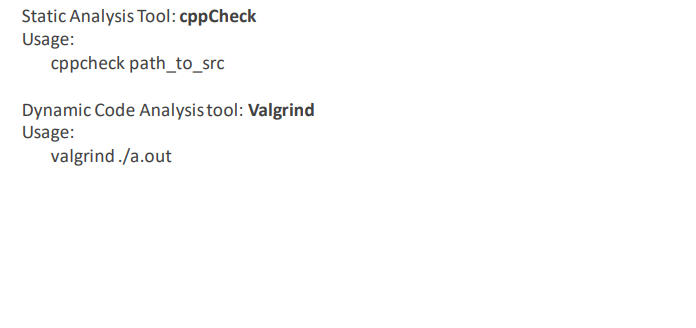
# Rules in Makefile:-



**GDB:-**

****

# Static and Dynamic Code analysis:-



**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](https://www.oreilly.com/library/view/linux-system-programming/9781449341527/ch01.html#c_language_standards)) and the system ABI (see [APIs and ABIs](https://www.oreilly.com/library/view/linux-system-programming/9781449341527/ch01.html#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](https://www.oreilly.com/library/view/linux-system-programming/9781449341527/ch03.html). The most important APIs in Linux system programming are discussed in the section [Standards](https://www.oreilly.com/library/view/linux-system-programming/9781449341527/ch01.html#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

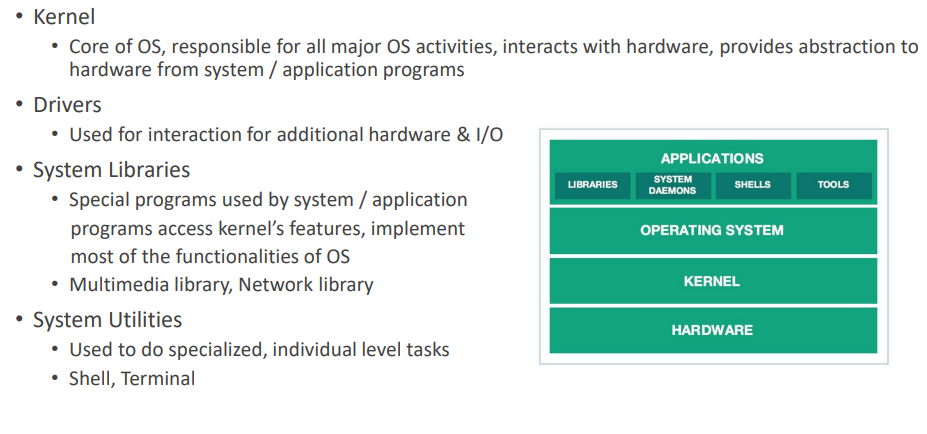
As stated earlier, Linux aims toward POSIX and SUS compliance. It provides the interfaces documented in SUSv4 and POSIX 2008, including real-time (POSIX.1b) and threading (POSIX.1c) support. More importantly, Linux strives to behave in accordance with POSIX and SUS requirements. In general, failing to 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](https://www.oreilly.com/library/view/linux-system-programming/9781449341527/apa.html).

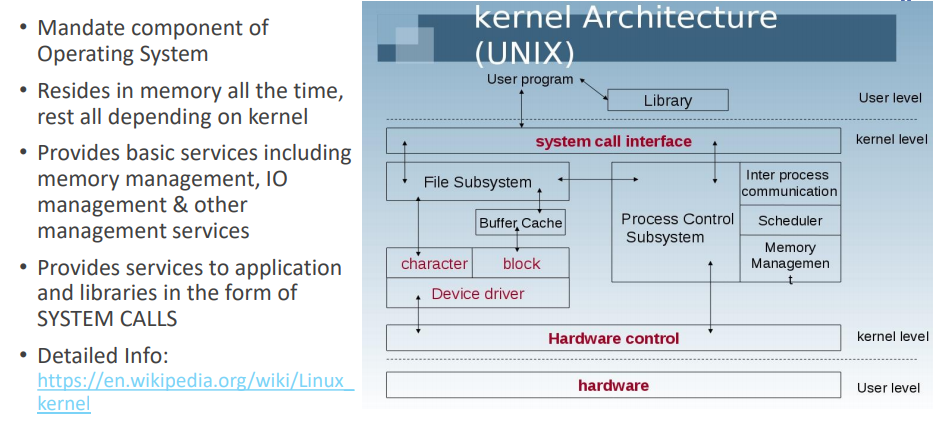
Linux has not had a great history of forward compatibility,[[1](https://www.oreilly.com/library/view/linux-system-programming/9781449341527/ch01.html#ftn.id683600)] although these days it fares much better. Interfaces documented by standards, such as the standard C library, will obviously always remain 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.

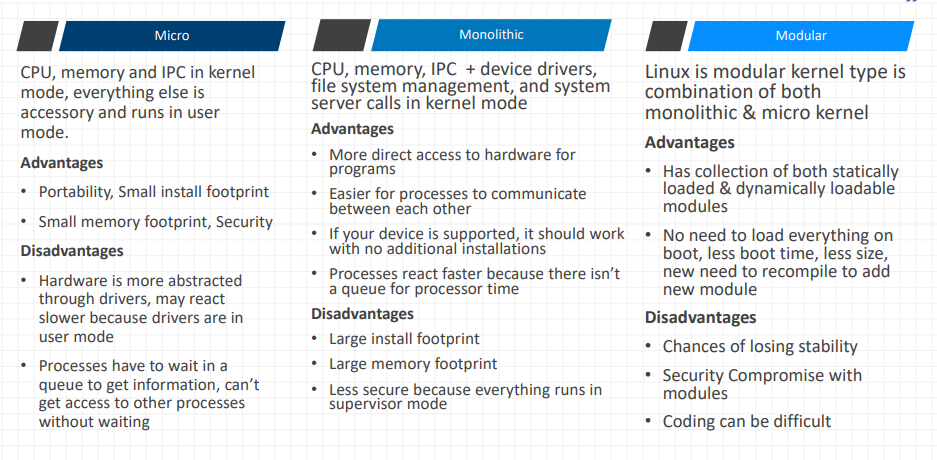
**Linux OS Architecture:-**

****

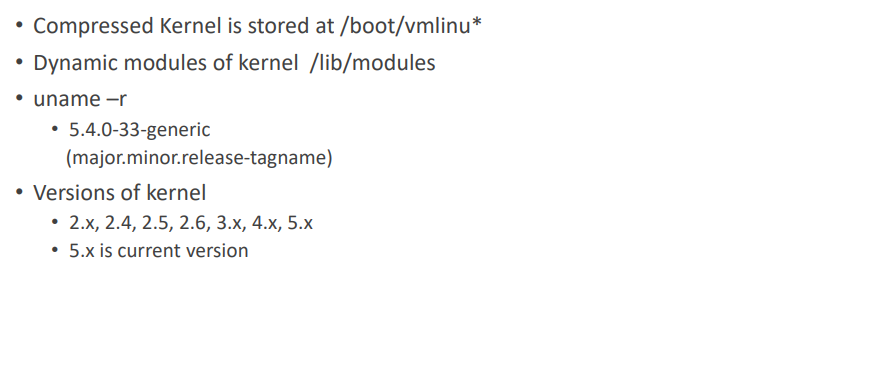
**Kernel:-**

****

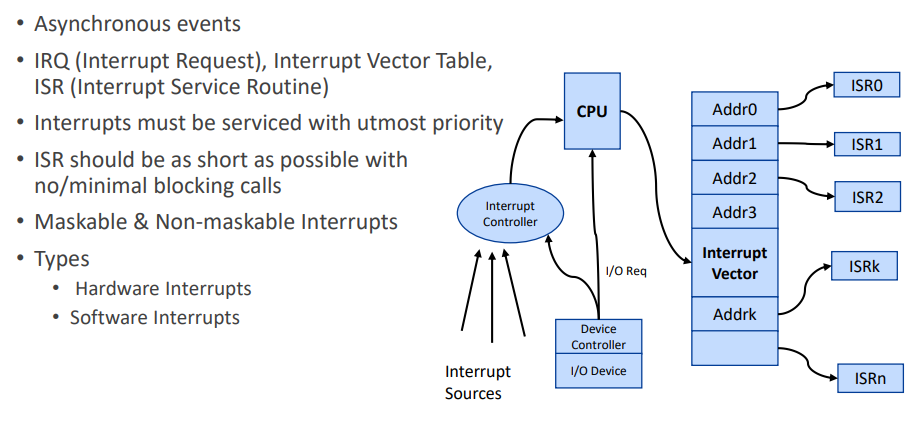
**Types of Kernel:-**

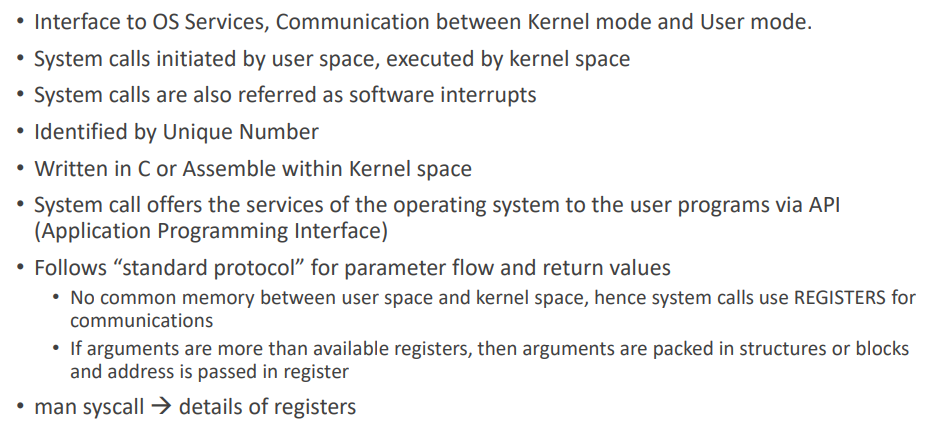
****

**Linux OS Kernel:-**

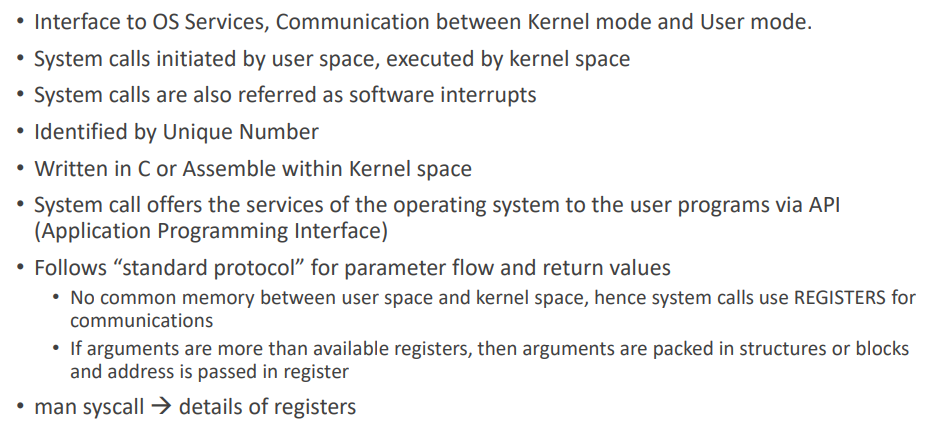
****

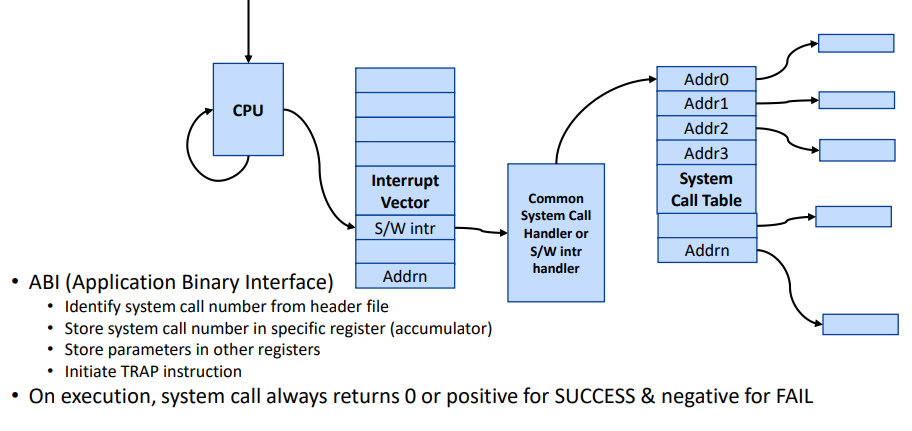
**Interrupts:-**

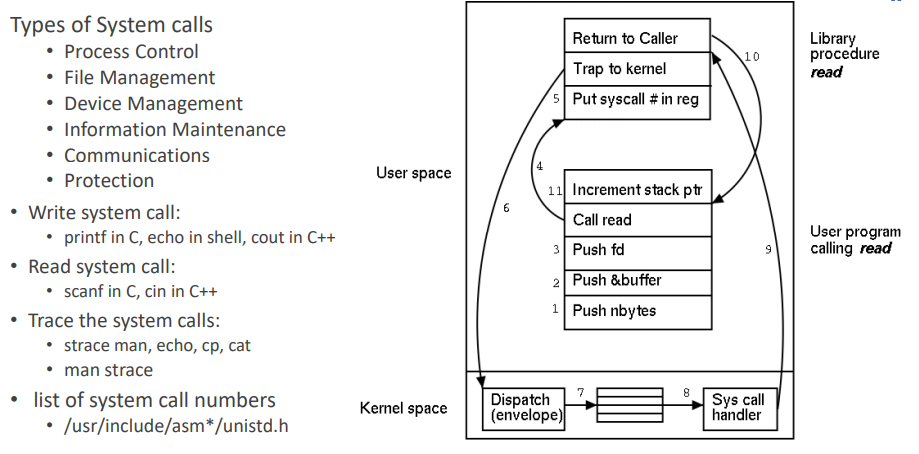
****

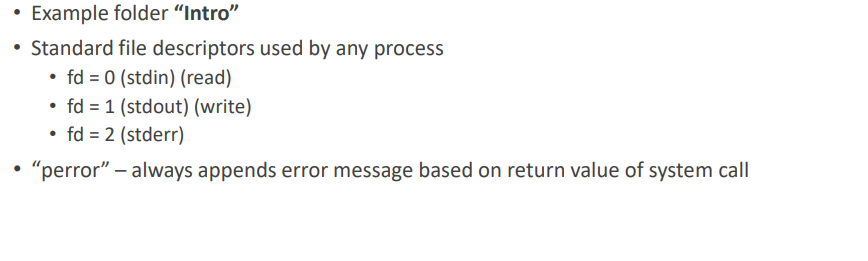


**System calls:-**

****

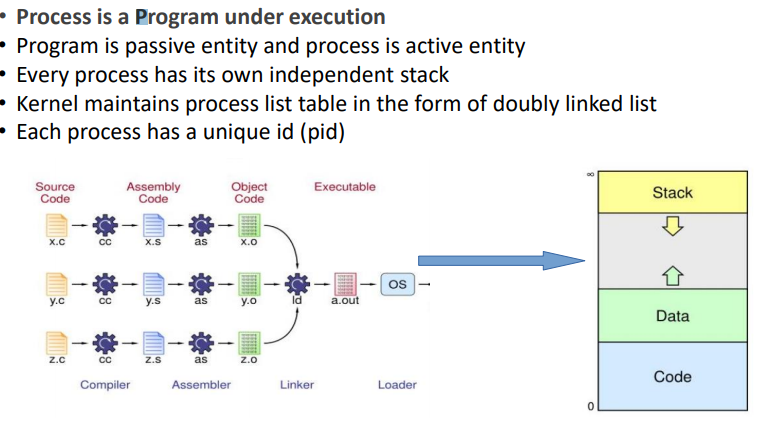
****

****

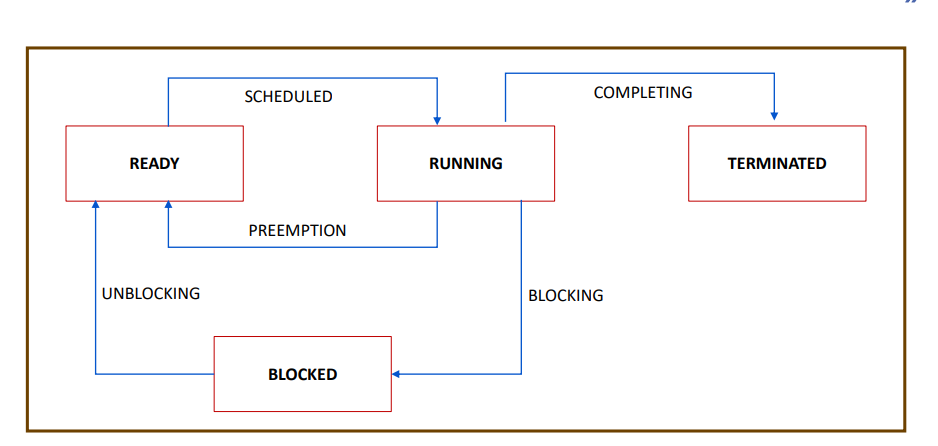
****

**Processs Management**

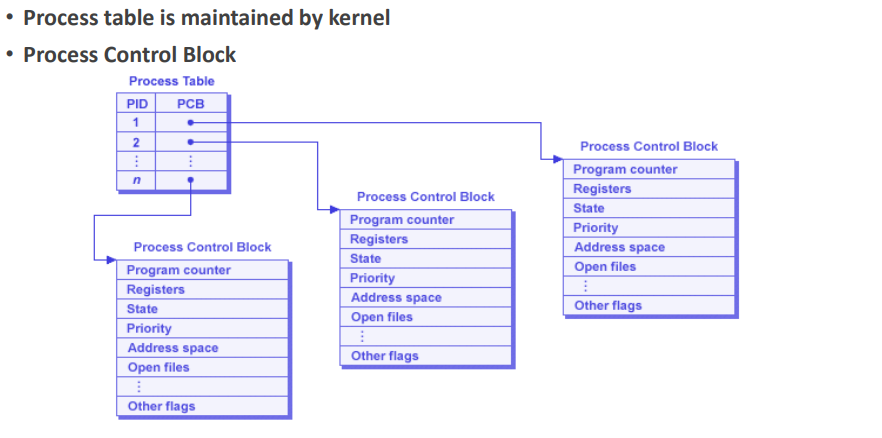
**Process:-**

****

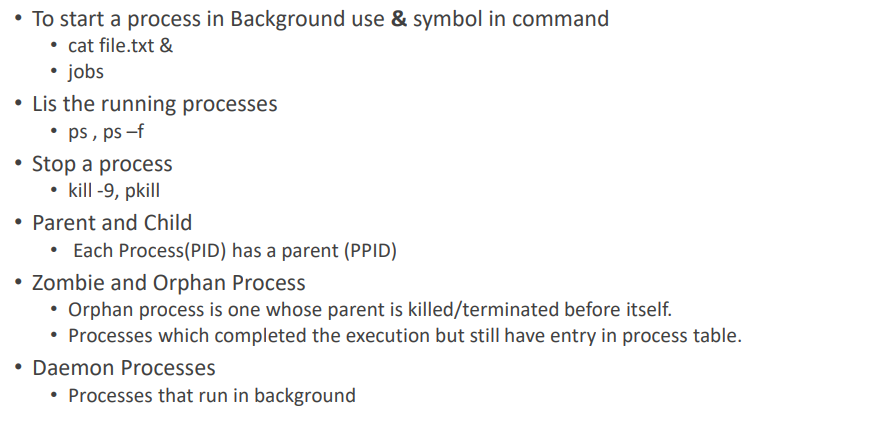
**Process Life Cycle:-**

****

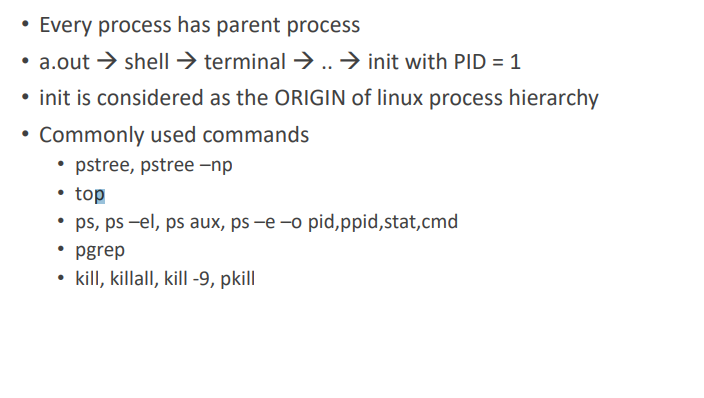
**Process Table and Process Control Block:-**

****

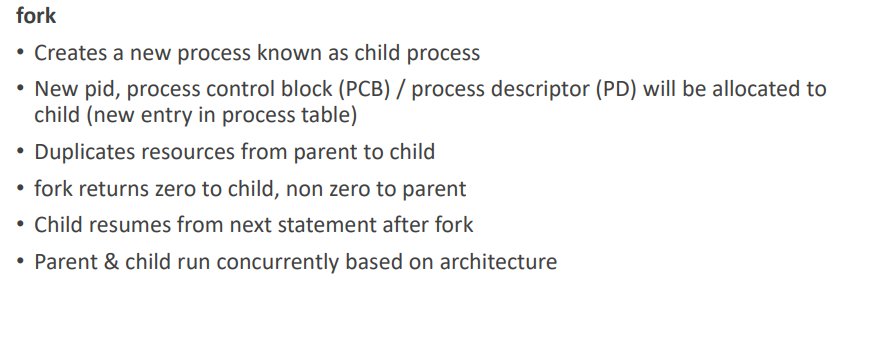
**Process:-**

****

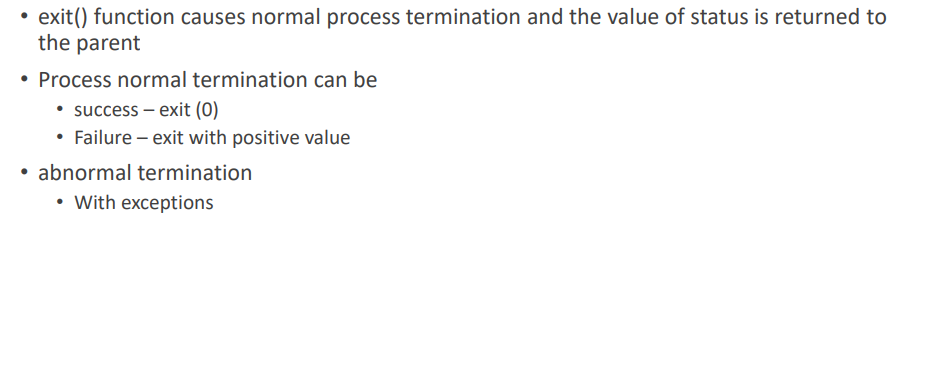
**Process Hierachy:-**

****

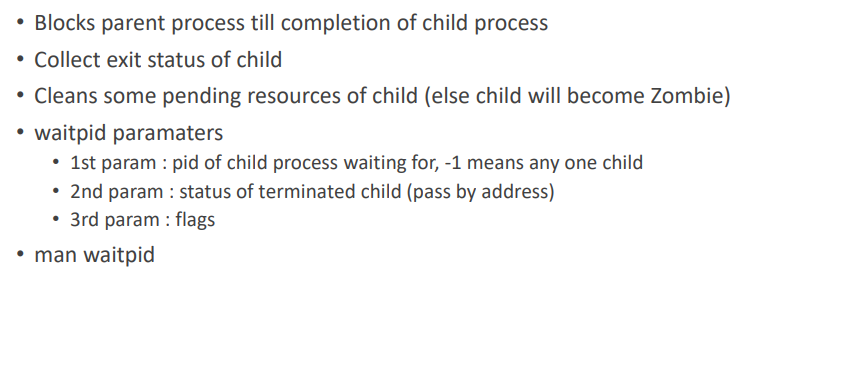
**New Process Creation:-**

****

**Process termination:-**

****

**Waitpid:-**

****