Case Study
on
Debian GNU/Linux

CSE315-4: Design of Operating System
Spring 2023

Azwad Fawad Hasan, 2020222
Md Rashid Shabab, 1910282
Tasnia Tabassum Oishee, 1910040

Table of Contents

Introduction

Nowadays, it is difficult to imagine everyday life without the use of computers. Every computer needs an operating system (OS) to work. An operating system is a set of basic system programs that manage computer hardware to perform basic computer functions, that is, it enables a connection between hardware and user programs. One of the most used systems today that contributed to the computer revolution is the Linux operating system named after its original author Linus Torvalds. The Linux Operating System became popular among computer users at the end of the 1990s. Many developing countries have adopted the Linux operating system for education and e-governance to avoid the financial burden of proprietary software. The advent of Ubuntu as a user-friendly Linux changed the history of Linux Desktop. With this, even home users started to install and use the Linux Operating System on desktop and laptop computers. Debian Linux project (https://www.debian.org/) is a community-owned project and follows the principles of Free and Open Source software (FOSS). Modern and popular Linux operating systems like Ubuntu derived from the Debian project ("Debian derivatives," 2018). The Debian project is unique among other Linux projects because of its extensive community support, large package repositories and free project management style. Besides, Debian Linux possesses a lot of useful features suitable for both new and expert user groups. The Debian project was initiated by Ian Murdock in August 1993. The rst release of the operating system rolled out in January 1994. The name Debian was coined by blending the rst name of his then-girlfriend Debra Lynn and the first name of Ian Murdock ("A Brief History of Debian," 2017). Three versions of Debian available; Old Stable, Stable and Experimental. The stable release of the operating system comes out every two years. The old Stable release is the previous version. The experimental release is the testing version of the Debian release. The Debian project follows both the numbering scheme and the code name for all releases. The code names are borrowed from the popular Toy Story series of films ("Debian releases and names," 2018). For example, Debian version 9 released in 2017 and its code name is 'Stretch'. Raul Silva designed the Debian 'swirl' logo in 1999. Many popular Linux distributions had derived from the Debian project. The popular Ubuntu Linux is inspired by the Debian project. As per the DistroWatch website, 143 active Debian derivatives("Debian," 2018). Debian package repositories contain 51000 application packages("Reasons to Choose Debian," 2017), which means that users can make available software for any purpose.

Free and Open Source software offers a lot of choices for users. User community should be aware of the features of the projects to enjoy the best of the Free and Open Source software. The Debian Linux project is one of the examples of community participation in operating system development. The project could successfully roll out stable Linux operating systems and provide long-term community support. The advantage of Debian is that it oers compatibility with a wide range of computing devices like a server, desktop, workstation, laptop, embedded system and mobile device.Debian Linux has not received much attention from the user community due to the lack of promotion.

Linux opens up more space for customization and innovation for experts and users. Debian follows a strict release schedule and ensures a stable distribution of Linux (Wallen, 2018).

Design goals/principles

Now we have some idea of what an OS actually does: it takes physical resources, such as a CPU, memory, or disk, and virtualizes them. It handles tough and tricky issues related to concurrency. And it stores files persistently, thus making them safe over the long-term. Given that we want to build such a system, we want to have some goals in mind to help focus our design and implementation and make trade-offs as necessary; finding the right set of trade-offs is a key to building systems. One of the most basic goals is to build up some abstractions in order to make the system convenient and easy to use. Abstractions are fundamental to everything we do in computer science. Abstraction makes it possible to write a large program by dividing it into small and understandable pieces, to write such a program in a high-level language like C without thinking about assembly, to write code in assembly without thinking about logic gates, and to build a processor out of gates without thinking too much about transistors.

One goal in designing and implementing an operating system is to provide high performance; another way to say this is our goal is to minimize the overheads of the OS. Virtualization and making the system easy to use are well worth it, but not at any cost; thus, we must strive to provide virtualization and other OS features without excessive overheads.

Another goal will be to provide protection between applications, as well as between the OS and applications. Because we wish to allow many programs to run at the same time, we want to make sure that the malicious or accidental bad behaviour of one does not harm others; we certainly don't want an application to be able to harm the OS itself (as that would affect all programs running on the system).

The operating system must also run non-stop; when it fails, all applications running on the system fail as well. Because of this dependence, operating systems often strive to provide a high degree of reliability. As operating systems grow ever more complex (sometimes containing millions of lines of code), building a reliable operating system is quite a challenge — and indeed, much of the on-going research in the field (including some of our own work) focuses on this exact problem.

Other goals make sense: energy-efficiency is important in our increasingly green world; security (an extension of protection, really) against malicious applications is critical, especially in these highly networked times; mobility is increasingly important as OSes are run on smaller and smaller devices. Depending on how the system is used, the OS will have different goals and thus likely be implemented in at least slightly different ways. However, as we will see, many of the principles we will present on how to build an OS are useful on a range of different devices.

Linux is a multiuser, multitasking system with a full set of UNIX-compatible tools.

Its file system adheres to traditional UNIX semantics, and it fully implements the standard UNIX networking model.

Linux is designed to be compliant with the relevant POSIX documents; at least two Linux distributions have achieved official POSIX certification.

The Linux programming interface adheres to the SVR4 UNIX semantics, rather than to BSD behaviour.

As PCs became more powerful and as memory and hard disks became cheaper, the original, minimalist Linux kernels grew to implement more UNIX functionality.

Speed and efficiency are still important design goals, but much recent and current work on Linux has concentrated on a third major design goal: standardisation. One of the prices paid for the diversity of UNIX implementations currently available is that source code written for one may not necessarily compile or run correctly on another.

Even when the same system calls are present on two different UNIX systems, they do not necessarily behave in exactly the same way.

Linux is designed to be compliant with the relevant POSIX documents; at least two Linux distributions have achieved official POSIX certification.

Operating-system structure

The structure of an Operating System determines how it has been designed and how it functions. There are numerous ways of designing a new structure of an Operating system. In this post, we will learn about six combinations that have been tested and tried. These six combinations are monolithic systems, layered systems, microkernels, client-server models, virtual machines, and exokernels.

Monolithic System structure in an Operating System

In this organisational structure, the entire operating system runs as a single program in the kernel mode. An operating system is a collection of various procedures linked together in a binary file. In this system, any procedure can call any other procedure. Since it is running in kernel mode itself, it has all the permissions to call whatever it wants.

In terms of information hiding, there is none. All procedures are running in kernel mode, so they have access to all modules and packages of other procedures.

However, using this approach without any restrictions can lead to thousands of procedure calls, and this can lead to a messy system. For this purpose, the actual OS is constructed in a hierarchy. All the individual procedures are compiled into a single executable file using the system linker.

Even a monolithic system has a structure in which it can run in user mode. There already is a basic structure given by the organisation

1. The main procedure that invokes the requested service procedures.
2. A set of service procedures that carry out system calls.
3. A set of utility procedures that help out the system procedures.

Layered Systems Structure in Operating Systems

As the name suggests, this system works in layers. It was designed by E.W. Dijkstra in 1968, along with some help from his students. This system was first implemented in THE system built at the Technosphere Hogeschool Eindhoven in the Netherlands. The system was a simple batch system for a Dutch computer, the Extralegal X8.

Working

There are six layers in the system, each with different purposes. Layer 0 – Processor Allocation and Multiprogramming – This layer deals with the allocation of processors, switching between the processes when interrupts occur or when the timers expire.

The sequential processes can be programmed individually without having to worry about other processes running on the processor. That is, layer 0 provides that basic multiprogramming of the CPU

Layer 1 – Memory and Drum Management – This layer deals with allocating memory to the processes in the main memory. The drum is used to hold parts of the processes (pages) for which space couldn’t be provided in the main memory. The processes don’t have to worry if there is available memory or not as layer 1 software takes care of adding pages wherever necessary.

Layer 2 – Operator-Process communication – In this layer, each process communicates with the operator (user) through the console. Each process has its own operator console and can directly communicate with the operator.

Layer 3 – Input/Output Management – This layer handles and manages all the I/O devices, and it buffers the information streams that are made available to it. Each process can communicate directly with the abstract I/O devices with all of its properties.

Layer 4 – User Programs – The programs used by the user are operated in this layer, and they don't have to worry about I/O management, operator/processes communication, memory management, or the processor allocation.

Layer 5 – The Operator – The system operator process is located in the outer most layer.

Microkernels system in an operating system

Traditionally, all the layers of the OS in a layered system went into the kernel. So they all had root access to the OS, and any small bug in any layer could be fatal to the OS.

Famous examples of a microkernel system include Integrity, K42, PikeOS, Symbian, and MINIX 3

The primary purpose of this system is to provide high reliability. Because of the high reliability that it provides, the applications of microkernels can be seen in real-time, industrial, avionics (electronics fitted in aircraft and aviation), and military applications that are mission-critical and require high reliability.

Working

The operating system is split into small, well-defined modules, of which only one, the microkernel, runs in kernel mode. The rest of the modules run as powerless ordinary user processes. Running each device driver and file system as separate user processes is a fail-safe method as a bug in one of the drivers will fail only that component. A bug easily references an invalid memory address and brings the system to a grinding halt instantly.

MINIX 3

Taking the example of MINIX 3 will help us understand microkernels much better.

The MINIX 3 is an OS written in C with 3200 lines of code and about 800 lines of code for the assembler to handle low-level functions like catching interrupts or switching processes. The C code handles managing and scheduling processes, handles interprocess communication. It also provides a set of 35 kernel calls to allow the rest of the operating system to do its work. These calls perform functions like hooking handlers to interrupts, moving data between address spaces, and installing new memory maps for newly created processes.

The process structure of MINIX 3 is divided into three parts above the kernel. The lowest layer contains device drivers. The middle layer includes servers. The uppermost layer contains user programs.

Client-Server Model in Operating Systems

The client-server model in an operating system is a variation of the microkernel system. The middle layer in the microkernel system is the one with servers. These servers provide some kind of service to clients. This makes up the client-server model.

Communication between clients and servers is obtained by message passing. To receive a service, one of the client processes constructs a message saying what it wants and sends it to the appropriate service. The service then does it work and sends back the answer.

If the clients and servers are on the same machine, then some optimizations are possible. But generally speaking, they are on different systems and are connected via a network link like LAN or WAN.

The best example of this model is you reading this article learning about it right now. You are the client, and you are requesting this page from whatever host this article has been uploaded to. The internet is basically the example since much of the web operates this way.

Virtual Machines in Operating systems

When many users wanted to work interactively in terminals, IBM started working on a time-sharing system. The idea of a virtual machine is straightforward. The virtual machine is run on the hardware of the OS it is being installed. A virtual machine thinks it has its own disk, with blocks running from 0 to some maximum, so the virtual machine monitor must maintain tables to remap disk addresses and all other resources.

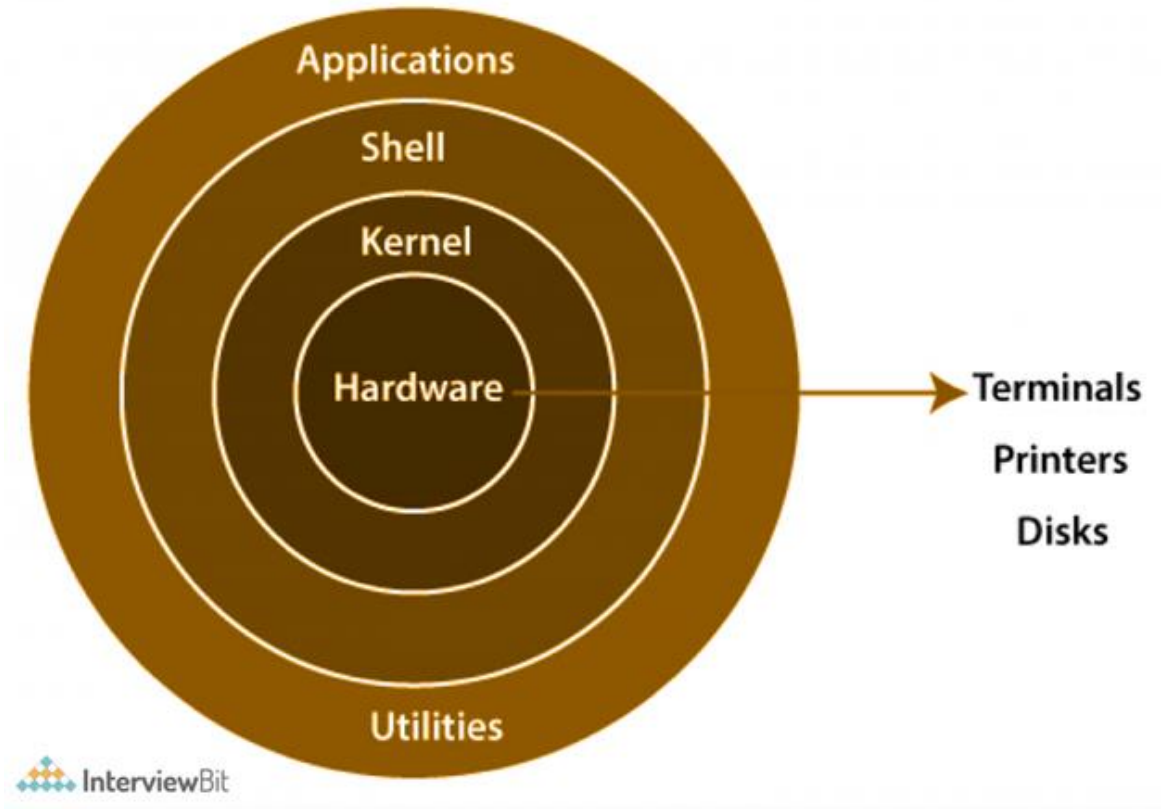
Exokernels

Exokernels are a subset of virtual machines. In this, the disks are actually partitioned, and resources are allocated while setting it up. The different OS may be installed on different partitions. Beneath both, the partition is what we call the exokernel.

Components of Operating system

A computer’s operating system interface to the hardware is referred to as a software application. A number of software applications are run on operating systems to manage hardware resources on a computer.

The diagram illustrates the structure of the Linux system, according to the layers concept.

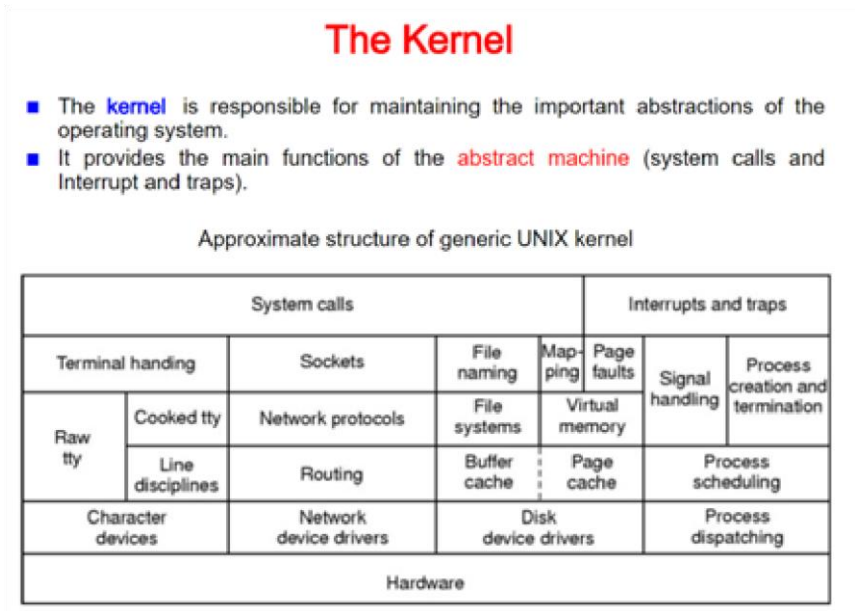


The Linux architecture is largely composed of elements such as the Kernel, System Library, Hardware layer, System, and Shell functions.

Kernel:

The kernel is one of the fundamental parts of an operating system. It is responsible for each of the primary duties of the Linux OS. Each of the major procedures of Linux is coordinated with hardware directly. The kernel is in charge of creating an appropriate abstraction for concealing trivial hardware or application strategies. The following kernel varieties are mentioned:

1. Monolithic Kernel
2. Micro kernels
3. Exo kernels
4. Hybrid kernels



The Kernel

- The kernel is responsible for maintaining the important abstractions of the operating system.
- It provides the main functions of the abstract machine (system calls and Interrupt and traps).

Approximate structure of generic UNIX kernel

System calls				Interrupts and traps		
Terminal handing		Sockets	File naming	Map-ping	Page faults	Signal handling	Process creation and termination
Raw tty	Cooked tty	Network protocols	File systems	Virtual memory		
	Line disciplines	Routing	Buffer cache	Page cache	Process scheduling	
Character devices		Network device drivers	Disk device drivers		Process dispatching	
Hardware						

System Libraries:

A set of library functions may be specified as these functions. These functions are implemented by the operating system and do not require code access rights on the kernel modules.

The Standard Library

- A system call is the method that the user process uses to ask for an action from the O.S.
- The programs perform the system calls by mean of **trap**.
  trap instruction:
  - changes from *user mode* to *kernel mode*
  - controls the correctness of the call parameters
  - execution done on behalf of the operating system
  - returns to user mode
- Since it is impossible to write a trap in C, it is provided a standard library with a procedure for each system call. These procedures are written in assembler and are called from C. For example a C program for performing the system call `read`, it calls the procedure *read* ,of the standard library.
- So, the standard library defines a standard set of functions through which applications interact with the kernel, and which implement much of the operating-system functionality that does not need the full privileges of kernel code.
- POSIX establishes which are the procedures of the library that the system has to provide, their parameters and tasks.

1.9

System Utility Programs:

A system utility program performs specific and individual jobs.

UNIX Utility Programs

Program	Typical use
cat	Concatenate multiple files to standard output
chmod	Change file protection mode
cp	Copy one or more files
cut	Cut columns of text from a file
grep	Search a file for some pattern
head	Extract the first lines of a file
ls	List directory
make	Compile files to build a binary
mkdir	Make a directory
od	Octal dump a file
paste	Paste columns of text into a file
pr	Format a file for printing
rm	Remove one or more files
rmdir	Remove a directory
sort	Sort a file of lines alphabetically
tail	Extract the last lines of a file
tr	Translate between character sets

A few of the more common UNIX utility programs required by POSIX

Hardware layer:

The hardware layer of Linux is made up of several peripheral devices such as a CPU, HDD, and RAM.

Shell:

Different operating systems are classified as graphical shells and command-line shells. A graphical shell is an interface between the kernel and the user. It provides kernel services, and it runs kernel operations. There are two types of graphical shells, which differ in appearance. These operating systems are divided into two categories, which are the graphical shells and command-line shells.

The graphical line shells allow for graphical user interfaces, while the command line shells enable for command line interfaces. As a result, both of these shells operate. However, graphical user interfaces performed using the graphical line shells are faster than those using the

The shell of a UNIX System

- The UNIX systems have a Graphical User Interface (Linux uses KDE, GNOME …), but the programmers prefer to type the commands.
- **Shell:** the user process which executes programs (command interpreter)
  - User types command
  - Shell reads command (read from input) and translates it to the operating system.
- Can run *external programs* (e.g. `netscape`) or *internal shell commands* (e.g. `cd`)
- Various different shells available:
  - Bourne shell (sh), C shell (csh), Korn shell (ksh), TC shell (tcsh), Bourne Again shell (bash).
- The administrator of the system provides to the user a default shell, but the user can change shell.

command line shells.

The shell of a UNIX System

- Example of commands
  - $ cd /usr/src/linux
  - $ more COPYING
  - $ cp file1 file2
  - $ head –20 file
  - $ head 20 file
  - $ ls *.c
  - $ ls [abc]*
  - $ sort < file1 > file2
- String multiple commands together in a shell script
  - $ sort < file1 > file2; head –30 < file2 ; rm temp
  - $ sort < file | head –30
  - $ grep org CREDITS | sort | head –20 | tail –5 > file

The shell is more …

- The shell is programmable, that is it possible to make the shell scripts
- A shell script is a file containing a sequence of commands addressed to the operating system that facilitates the repeated execution of the included commands without their having to be laboriously retyped each time they are executed.
- If there is a distinct ordered list of operating system commands that the user needs to execute repeatedly, for example, immediately after every login or immediately before every logout, then most operating systems have a facility for recording the list of commands in a file, which can then either be executed automatically upon login or logout, or can be invoked by the user through the issuance of a single command that results in the execution of the entire contents of the batch file, which can contain as few as one operating system command or as many as thousands.

1.6

The shell scripts

- Example:
  - ```
    #!/bin/csh
    clear
    echo Menuset
    stop=1
    while ($stop>0) cat << ENDOFMENU
            1: stampa data
            2: stampa la directory corrente
            3: esci
            ENDOFMENU
    echo -n Scegli il comando
    set reply=$<
    switch ($reply)
    case 1 :        date
                    breaksw
    case 2 :        pwd
                    breaksw
    case 3 :        set stop=0
                    breaksw
    default:        echo scelta sbagliata riprova
                    breaksw
    endsw
    end
    exit 0
    ```


Shells of the operating system

Shells are an important part of any Linux user session. We are provided several different types of shells in Linux to accomplish tasks. Each shell has unique properties. Hence, there are many instances where one shell is better than the other for specific requirements.

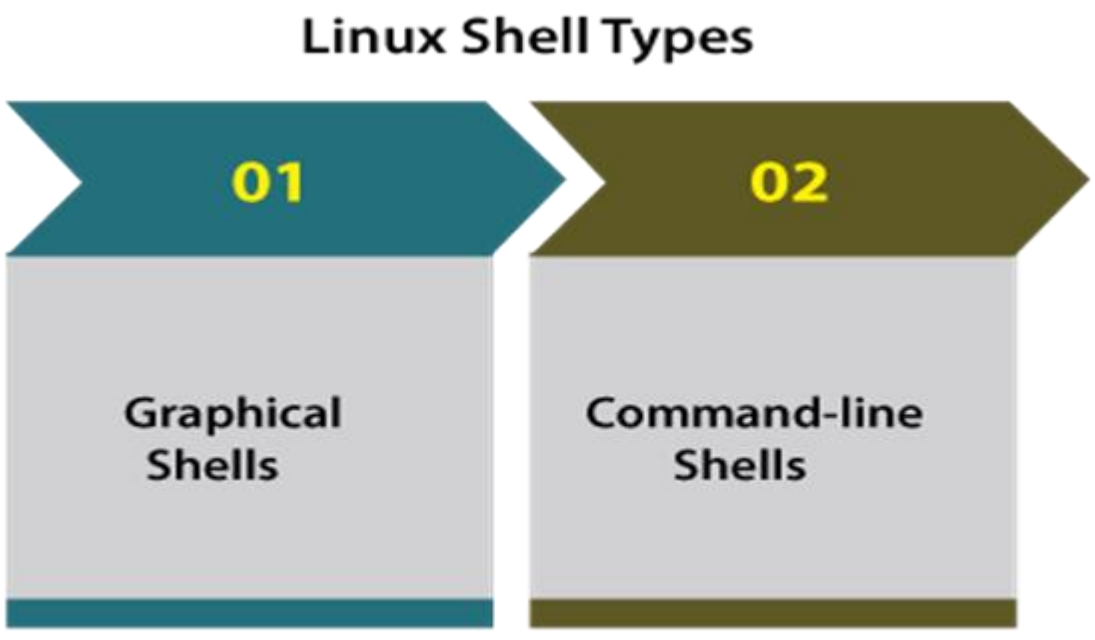
Whenever a user logs in to the system or opens a console window, the kernel runs a new shell instance. The kernel is the heart of any operating system.

It is responsible for the control management, and execution of processes, and to ensure proper utilisation of system resources.

A shell is a program that acts as an interface between a user and the kernel. It allows a user to give commands to the kernel and receive responses from it. Through a shell, we can execute programs and utilities on the kernel. Hence, at its core, a shell is a program used to execute other programs on our system. Being able to interact with the kernel makes shells a powerful tool. Without the ability to interact with the kernel, a user cannot access the utilities offered by their machine’s operating system.

Each of these shells has properties that make them highly efficient for a specific type of use over other shells. So let us discuss the different types of shells in Linux along with their properties and features.

Broadly, the shell is categorised into two main categories which are explained below:



Graphical Shells

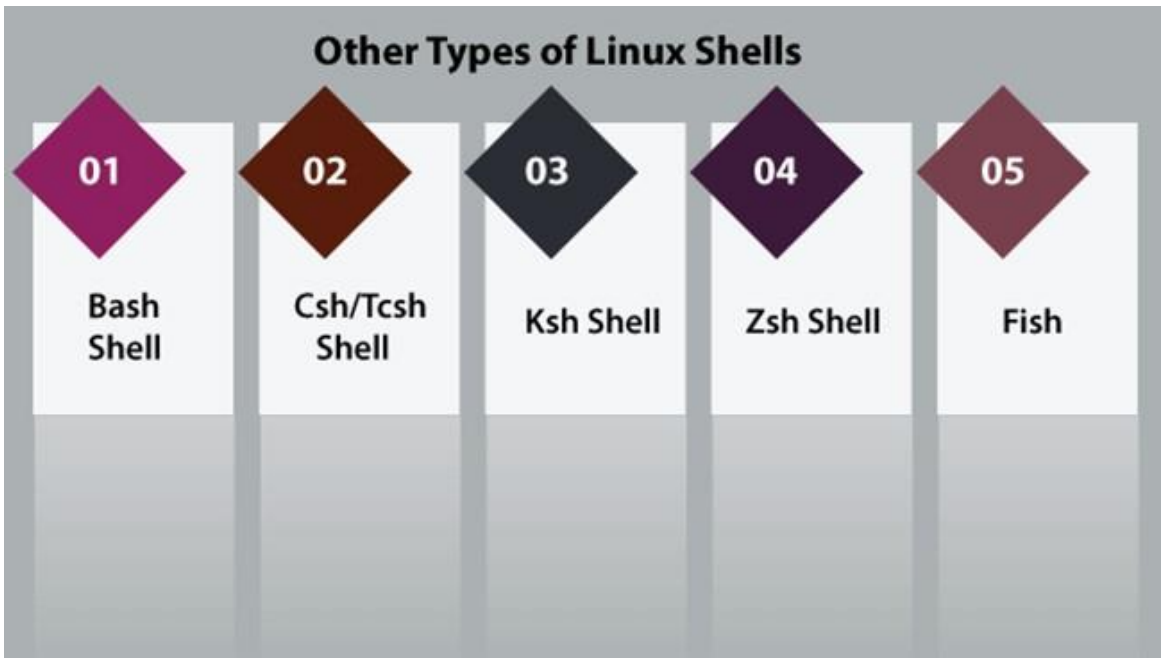
These shells specify the manipulation of programs that are based on the graphical user interface (GUI) by permitting for operations like moving, closing, resizing, and opening windows and switching focus among windows as well. Ubuntu OS or Windows OS could be examined as a good example that offers a graphical user interface to the user to interact with the program. Various users don't need to type in any command for all the actions.
    ```

Command-line Shell

Various shells could be accessed with the help of a command-line interface by users. A unique program known as Command prompt in Windows or Terminal in macOS/Linux is offered for typing in the human-understandable commands like "ls", "cat", etc and after that, it is being run. The result is further shown to the user on the terminal.

Working on a command-line shell is complicated for many beginners due to it being hard to remember several commands. Command-line shell is very dominant and it permits users to store commands in a file and run them together. In this way, a repetitive action could be automated easily. Usually, these files are known as Shell scripts in macOS/Linux systems and batch files in Windows.

There are various types of shells which are discussed as follows:



Bash Shell

In the bash shell, bash means Bourne Again Shell. It is a default shell over several distributions of Linux today. It is a sh-compatible shell. It could be installed over Windows OS. It facilitates practical improvements on sh for interactive and programming use which contains: · Job Control

- Command-line editing
- Shell Aliases and Functions
- Unlimited size command history
- Integer arithmetic in a base from 2-64

Csh/Tcsh Shell

Tcsh is an upgraded C shell. This shell can be used as a shell script command processor and interactive login shell. Tcsh shell includes the following characteristics:

- C like syntax
- Filename completion and programmable word
- Command-line editor
- Job control
- Spelling correction

Ksh Shell

Ksh means Korn shell. It was developed and designed by David G. Korn. Ksh shell is a high-level, powerful, and complete programming language and it is a reciprocal command language as well just like various other GNU/Unix Linux shells. The usage and syntax of the C shell are very similar to the C programming language.

Zsh Shell

Zsh shell is developed to be reciprocal and it combines various aspects of other GNU/Unix Linux shells like ksh, tcsh, and bash. Also, the POSIX shell standard specifications were based on the Korn shell.

Also, it is a strong scripting language like other available shells. Some of its unique features are listed as follows:

- Startup files
- Filename generation
- Login/Logout watching
- Concept index
- Closing comments
- Variable index
- Key index
- Function index and various others that we could find out within the man pages.

All these shells do a similar job but take different commands and facilitate distinct built-in functions.

Fish

Fish stands for "friendly interactive shell". It was produced in 2005. Fish shell was developed to be fully user-friendly and interactive just like other shells. It contains some good features which are mentioned below:

- Web-based configuration
- Man page completions
- Auto-suggestions
- Support for term256 terminal automation
- Completely scripted with clean scripts

Shell Prompt

It is known as a command prompt and it is issued via the shell. We can type any command while the prompt is shown.

Shell reads our input after we click Enter. It illustrates the command we want to be run by seeing at the initial word of our input. A word can be defined as the character's unbroken set. Tabs and spaces separate words.

The below is a common data command example that shows the current time and date:

We can also customise our command prompt with the help of PS1 (environment variable).

Shell Scripting

The common concept of the shell script is the command list. A good shell script will contain comments which are preceded via # simbol.

Shell functions and scripts are interpreted. It means they aren't compiled.

There are also conditional tests like value Y is greater than value Z, loops permitting us to proceed by massive data amounts, files to store and read data, and variables to store and read data, and these scripts may contain functions.

The shells are usually interactive which means they receive commands as input through the users and run them. Although sometimes we wish to run a set of commands, hence, we have to type the commands all-time inside the terminal.

A shell script includes syntax similarly to other programming languages. When we have prior experience along with a programming language such as C/C++, Python, etc. It will be very easy to begin with. The shell script combines the below components:

Functions

- Control flow: if, else, then, shell loops, case, etc.
- Shell commands: touch, pwd, echo, ls, cd, etc.
- Shell keywords: break, if, else, etc.

Shell Scripts Need

There are several causes to write these shell scripts:

- For avoiding automation and repetitive work
- Shell scripting is used by system admins for many routine backups
- Including new functionalities to the

shell · System monitoring etc.

Shell Script Advantages

- The syntax and command are exactly similar to those entered directly in a command line. Thus, the programmers don't have to switch to completely different syntax
- Interactive debugging
- Quick start
- It is much quicker to write the shell scripts etc.

Shell Script Disadvantages

- A single error can modify the command which could be harmful, so prone to very costly errors.
- Design flaws in the language implementation and syntax.
- Slower execution speed.
- Offer minimal data structure dissimilar to other scripting languages.
- Not well suited for complex and large tasks etc.

Programmer Interface

There are two major parts of the programming interface: one seen by developers of applications, which is the API, and the other one is seen by developers of system components, such as device drivers and platform support modules, which is the SPI (system programming interface).

The Linux API is the kernel–user space API, which allows programs in user space to access system resources and services of the Linux kernel. It is composed out of the System Call Interface of the Linux kernel and the subroutines in the GNU C Library (glibc). The focus of the development of the Linux API has been to provide the usable features of the specifications defined in POSIX in a way which is reasonably compatible, robust and performant, and to provide additional useful features not defined in POSIX, just as the kernel– user space APIs of other systems implementing the POSIX . API also provides additional features not defined in POSIX.

Process creation, termination and communication

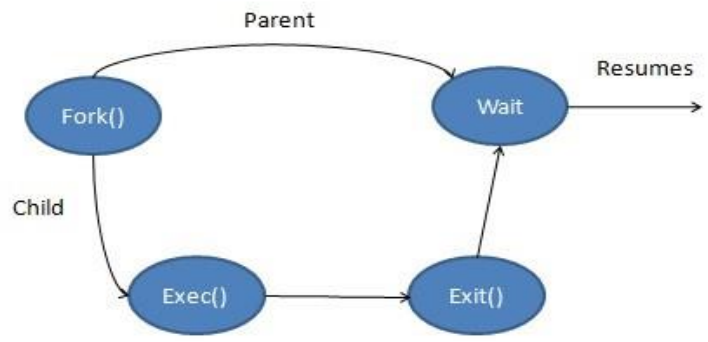
PROCESS CREATION:

A process can create a new process using the fork() system call. The process that calls fork() is referred to as the parent process, and the new process is referred to as the child process. The kernel creates the child process by making a duplicate of the parent process. The child inherits copies of the parent's data, stack, and heap segments, which it may then modify independently of the parent's copies. The child process goes on either to execute a different set of functions in the same code as the parent, or, frequently, to use the execve() system call to load and execute an entirely new program. An execve() call destroys the existing text, data, stack, and heap segments, replacing them with new segments based on the code of the new program.

Each process has a unique integer process identifier (PID). Each process also has a parent process identifier (PPID) attribute, which identifies the process that requested the kernel to create this process.

PROCESS TERMINATION:

A process can terminate in one of two ways: by requesting its own termination using the exit system call, or by being killed by the delivery of a signal. In either case, the process yields a termination status, a small non-negative integer value that is available for inspection by the parent process using the wait system call. In the case of a call to exit, the process explicitly specifies its own termination status. If a process is killed by a signal, the termination status is set according to the type of signal that caused the death of the process. (Sometimes, it is referred to as the argument passed to _exit() as the exit status of the process, as distinct from the termination status, which is either the value passed to _exit() or an indication of the signal that killed the process). By convention, a termination status of 0 indicates that the process succeeded, and a nonzero status indicates that some error occurred.



PROCESS COMMUNICATION:

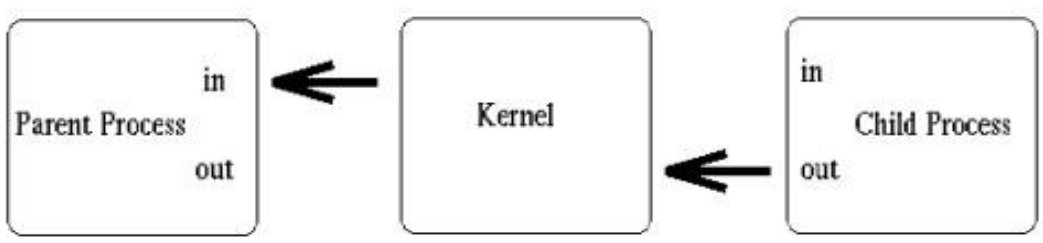
Inter-Process Communication (IPC) refers to a mechanism, where the operating systems allow various processes to communicate with each other. This involves synchronizing their actions and managing shared data.

There are several reasons for providing an environment that allows process cooperation:

- Information sharing.
- Speedup.
- Modularity.
- Convenience.

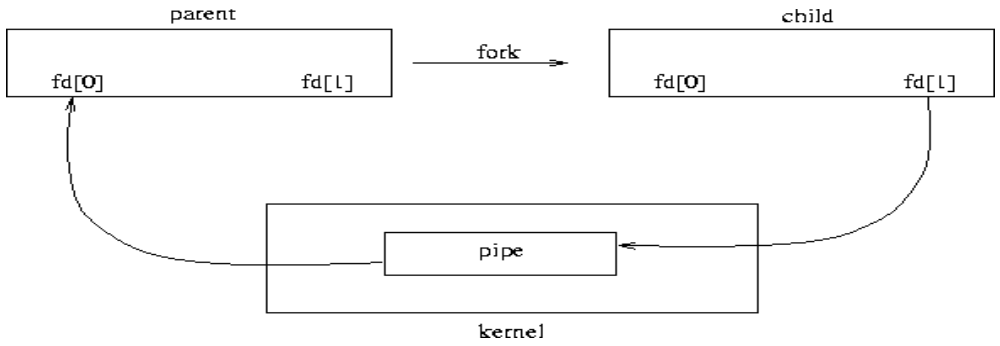
The Linux IPC (Inter-process communication) facilities provide a method for multiple processes to communicate with one another.

- **Half-duplex UNIX pipes:**
  - A pipe is a method of connecting the standard output of one process to the standard input of another. Pipes are the eldest of the IPC tools, having been around since the earliest incarnations of the UNIX operating system. They provide a method of one-way communications (hence the term half-duplex) between processes.
  - This feature is widely used, even on the UNIX command line (in the shell). **ls | sort | lp**
  - The above sets up a pipeline, taking the output of ls as the input of sort, and the output of sort as the input of lp. The data is running through a half-duplex pipe, travelling (visually) left to right through the pipeline.
  - When a process creates a pipe, the kernel sets up two file descriptors for use by the pipe. One descriptor is used to allow a path of input into the pipe (write), while the other is used to obtain data from the pipe (read).
  - To send data to the pipe, the write() system call is used, and to retrieve data from the pipe, the read() system call is used.

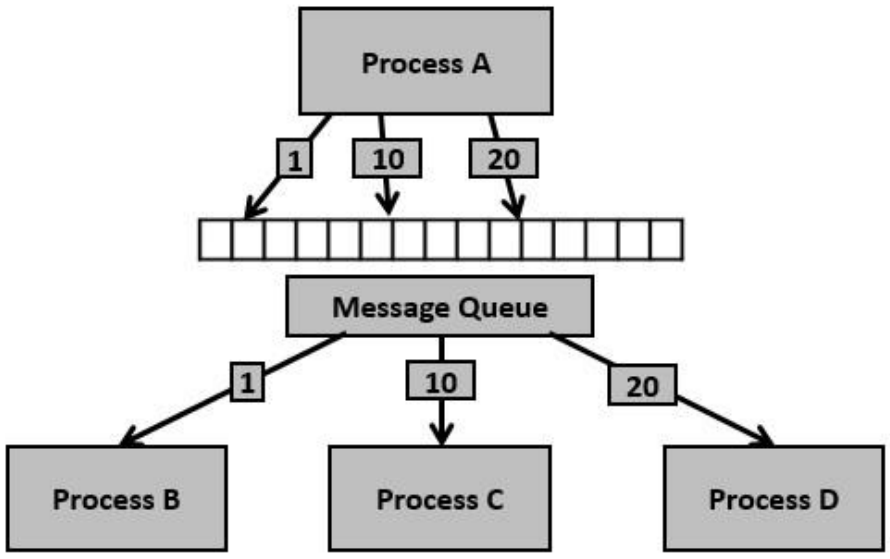


- **FIFOs (named pipes):**

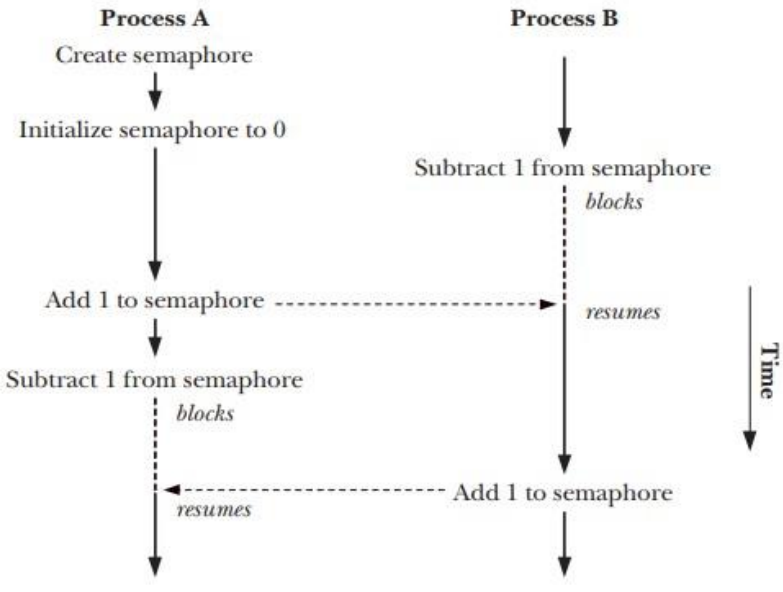
- A named pipe works much like a regular pipe but does have some noticeable differences. ○ Named pipes exist as a device special file in the file system. ○ Processes of different ancestry can share data through a named pipe. ○ When all I/O is done by sharing processes, the named pipe remains in the file system for later use.



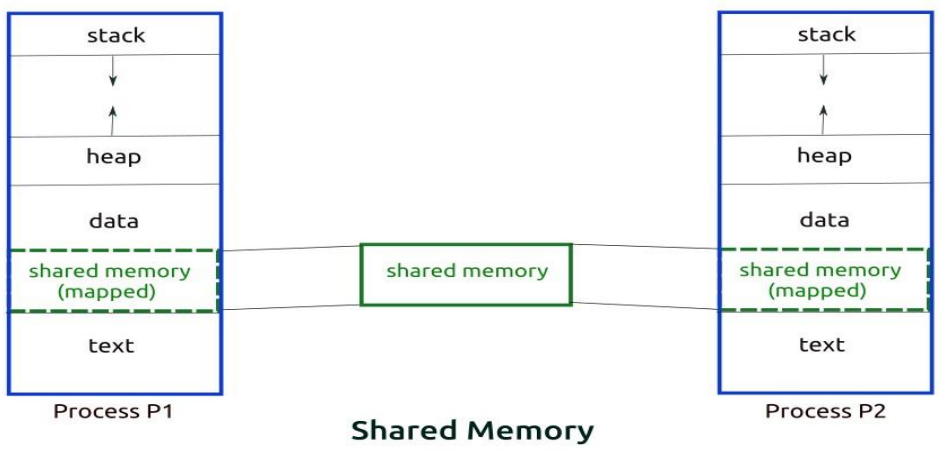
- SYSV style message queues:
  - Message queues can be best described as an internal linked list within the kernel's addressing space. Messages can be sent to the queue in order and retrieved from the queue in several different ways. Each message queue (of course) is uniquely identified by an IPC identifier.
  - Pipes have strict FIFO behavior: the first byte written is the first byte read, the second byte written is the second byte read, and so forth. Message queues can behave in the same way but are flexible enough that byte chunks can be retrieved out of FIFO order.



- **SYSV style semaphore sets:**
  - Semaphores can best be described as counters used to control access to shared resources by multiple processes. They are most often used as a locking mechanism to prevent processes from accessing a particular resource while another process is performing operations on it. In its simplest form a semaphore is a location in memory whose value can be tested and set by more than one process. The test and set operation are so far as each process is concerned, uninterruptible or atomic; once started nothing can stop it. The result of the test and set operation is the addition of the current value of the semaphore and the set value, which can be positive or negative. Depending on the result of the test and set operation one process may have to sleep until the semaphore's value is changed by another process. Semaphores can be used to implement critical regions, areas of critical code that only one process at a time should be executing.
  - There is a problem with semaphores, deadlocks. These occur when one process has altered the semaphores value as it enters a critical region but then fails to leave the critical region because it crashed or was killed. Linux protects against this by maintaining lists of adjustments to the semaphore arrays. The idea is that when these adjustments are applied, the semaphores will be put back to the state that they were in before a process's set of semaphore operations were applied.



- **SYSV style shared memory segments:**
  - In Shared memory, the mapping of an area (segment) of memory will be mapped and shared by more than one process. This is by far the fastest form of IPC, because there is no intermediation (i.e., a pipe, a message queue, etc.). Instead, information is mapped directly from a memory segment, and into the addressing space of the calling process. A segment can be created by one process, and subsequently written to and read from by any number of processes.



Process state diagram and process management system-calls

PROCESS STATES:

For any Linux process, their starting point is the moment they are created. Once it starts, the process goes into the running or runnable state. While the process is running, it could come into a code path that requires it to wait for resources or signals before proceeding. While waiting for the resources, the process would voluntarily give up the CPU cycles by going into one of the two sleeping states. Additionally, a running process could be suspended and put into a stopped state. A process in this state will continue to exist until it is killed or resumed. Finally, the process completes its life cycle when it's terminated and placed into a zombie state until its parent process clears it off the process table. Figure below shows the process state diagram of the operating system. Therefore, the possible states a process can be in is given below:

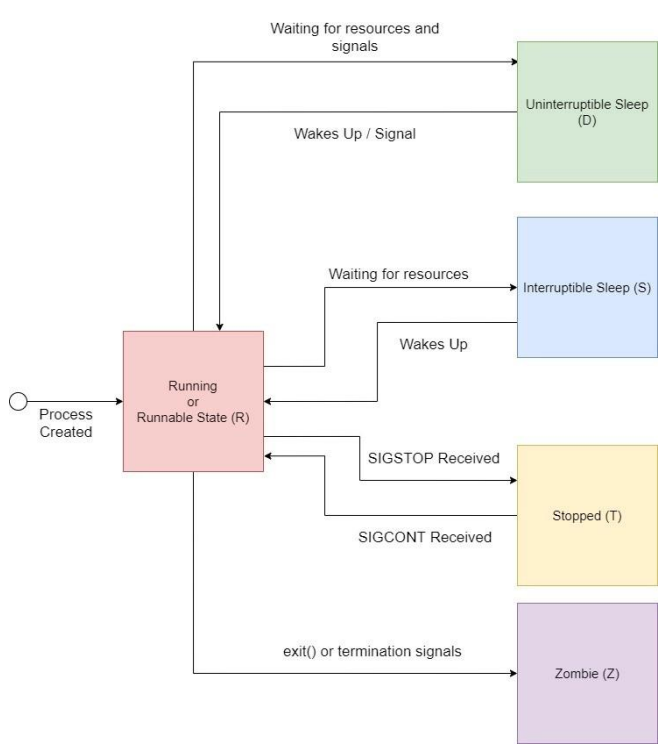
1.**RUNNING & RUNNABLE**: When the CPU executes a process, it will be in a RUNNING state. When the process is not waiting for any resource and ready to be executed by the CPU, it will be in the RUNNABLE state.

2.**INTERRRUPTABLE_SLEEP**: When a process is in INTERRUPTABLE_SLEEP, it will wake up from the middle of sleep and process new signals sent to it.

3.**UNINTERRUPTABLE_SLEEP**: When a process is in UNINTERRUPTABLE_SLEEP, it will not wake up from the middle of sleep even though new signals are sent to it.

4.**STOPPED**: STOPPED state indicates that the process has been suspended from proceeding further. In Linux when the 'Ctrl + Z' is issued, this command will issue a SIGSTOP signal to the process. When the process receives this signal, it will be suspended/stopped from executing further. When a process is in STOPPED state, it will only handle SIGKILL and SIGCONT signals. SIGKILL signal will terminate the process, but the SIGCONT signal will put the process back into RUNNING/RUNNABLE state.

5. **ZOMBIE**: A process will remain in this state until the parent process clears it off from the process table. To clear the terminated child process off the process table, the parent process must read the exit value of the child process using the wait() or wait PID() system calls.



There are multiple ways to check the state of a process in Linux. For example, we can use command-line tools like PS and top to check the state of processes. Alternatively, we can consult the pseudo status file for a particular PID.

PROCESS MANAGEMENT SYSTEM CALLS:

System call provides an interface between user program and operating system. The structure of system call is as follows −

When the user wants to give an instruction to the OS, then it will do it through system calls. Or a user program can access the kernel which is a part of the OS through system calls. It is a programmatic way in which a computer program requests a service from the kernel of the operating system. A system is used to create a new process, or a duplicate process called a fork.
The duplicate process consists of all data in the file description and registers common. The original process is also called the parent process and the duplicate is called the child process.
The fork call returns a value, which is zero in the child and equal to the child's PID (Process Identifier) in the parent. The system calls like exit would request the services for terminating a process.
Loading of programs or changing of the original image with duplicate needs execution of exec. PID would help to distinguish between child and parent processes.

Process management system calls in Linux.

- **fork()** − A parent process always uses a fork for creating a new child process. The child process is generally called a copy of the parent. After execution of fork, both parent and child execute the same program in separate processes.
- **exec()** − This function is used to replace the program executed by a process. The child sometimes may use exec after a fork for replacing the process memory space with a new program executable making the child execute a different program than the parent.
- **exit()** − This function is used to terminate the process.
- **wait()** − The parent uses a wait function to suspend execution till a child terminates. Using wait the parent can obtain the exit status of a terminated child.

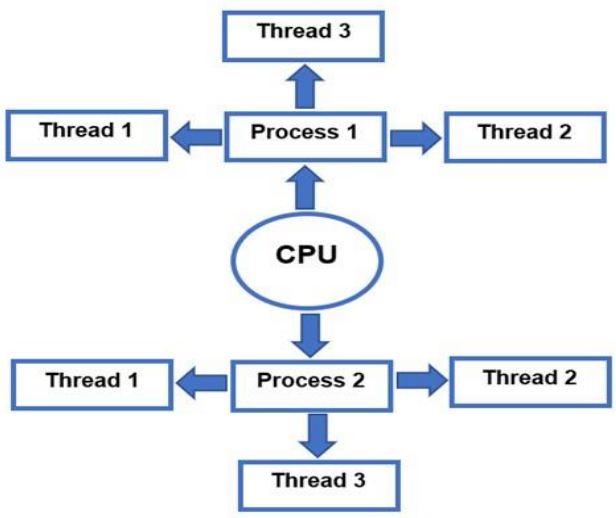
Process versus thread

Process:

A process is the execution of a program that allows it to perform the appropriate actions specified in a program. It can be defined as an execution unit where a program runs. The OS helps to create, schedule, and terminate the processes which are used by the CPU. The other processes created by the main process are called child processes.

Threads:

Thread is an execution unit that is part of a process. A process can have multiple threads, all executing at the same time. It is a unit of execution in concurrent programming.



Just as a process is identified through a process ID, a thread is identified by a thread ID. Following are some of the major differences between the thread and the processes:

- Processes do not share their address space while threads executing under the same process share the address space.
- Context switching between threads is fast as compared to context switching between processes
- The interaction between two processes is achieved only through the standard inter process communication while threads executing under the same process can communicate easily as they share most of the resources like memory, text segment etc.
- A process ID is unique across the system whereas a thread ID is unique only in the context of a single process.
- A process ID can be printed very easily while a thread ID is not easy to print.

Figure below shows further comparison between processes and threads in Linux.

Classification	Process	Threads
Definition	Execution of any program is a process	Segment or subset of a process is a thread
Communication	It takes more time to communicate between the different processes.	It takes less time for the communications
Resources	It consumes more resources	It consumes fewer resources
Memory	The process does not share the memory and is carried out alone	Threads are used to share their memory
Data Sharing	Process does not share their data	Threads share their data
Size	The process consumes more size	Threads are lightweight
Execution error	One process will not affect any other process due to an error	Threads will not run if one thread has an error
Termination time	Process usually takes more time to terminate	Threads take less time to terminate

User level thread and kernel level thread: models

Linux has a unique implementation of threads. There is no concept of a thread in the Linux kernel, where Linux implements all threads as standard processes. The Linux kernel does not provide any special scheduling semantics or data structures to represent threads. Instead, a thread is merely a process that shares certain resources with other processes. Basically, the normal fork() function is used to create threads. In Addition, users can use clone() to create threads as well. The two main types of threads are user-level threads and kernel-level threads.

USER-LEVEL THREADS:

The user-level threads are applied by the user through creating, controlling and destroying them using user space thread libraries. The kernel is not aware of the existence of these threads. It handles them as if they were single-threaded processes. User-level threads are small and much faster than kernel level threads. They are represented by a program counter (PC), stack, registers and a small process control block. Also, there is no kernel involvement in synchronisation for user-level threads.

The advantages of user space threads are that the switching between two threads does not involve much overhead and is generally very fast while on the negative side, since these threads follow cooperative multitasking so if one thread gets blocked the whole process gets blocked.

KERNEL-LEVEL THREADS:

In case of Kernel-level, threads are created, controlled and destroyed by the kernel. There is a corresponding kernel thread for every thread that exists in user space. Since these threads are managed by the kernel, they follow blocking multitasking where-in the scheduler can block a thread in execution with a higher priority thread which is ready for execution. The major advantage of kernel threads is that even if one of the threads gets blocked, the whole process is not blocked as kernel threads follow blocking scheduling while on the negative side, the context switch is not very fast as compared to user space threads.

In the case of Linux, it is task_struct. The only threads that are considered by the CPU scheduler for scheduling are Kernel threads. In Linux, kernel threads are optimized to such an extent that they are considered better than user space threads and mostly used in all scenarios except where prime requirement is that of cooperative multitasking.

Parameters	User Level	Kernel Level
Implemented by	User threads are implemented by users.	Kernel threads are implemented by Linux Debian.
Recognize	Linux Debian doesn’t recognize user level threads.	Kernel threads are recognized by Linux.
Implementation	Implementation of User threads is easy.	Implementation of Kernel thread is complicated.
Context switch time	Context switch time is less.	Context switch time is more.
Hardware support	Context switch requires no hardware support.	Hardware support is needed.

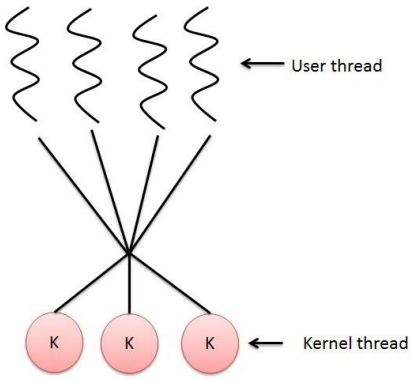
Blocking operation	If one user level thread performs blocking operation, then the entire process will be blocked.	If one kernel thread performs blocking operation, then another thread can continue execution.
Multithreading	Multi-threaded applications cannot take advantage of multiprocessing.	Kernels can be multi-threaded.

MULTI-THREADING MODELS:

- Many-to-many relationship.
- Many-to-one relationship.
- One-to-one relationship

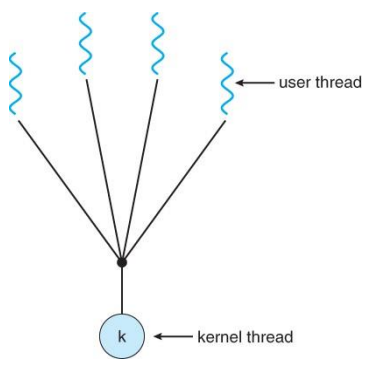
MANY-TO-MANY MODEL:

The many-to-many model complexes many user-level threads to a small or equal number of kernel threads. The number of kernel threads may be specific to either a particular application or a machine unlike many to one model.



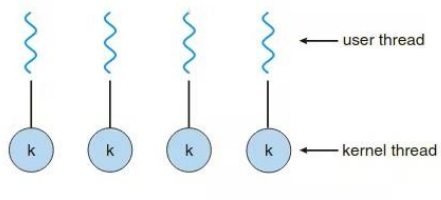
MANY-TO-ONE MODEL:

In case of many user-level, threads are all mapped onto a single kernel thread. Thread management is done in user space by the thread library. The entire process will be blocked once a thread makes a blocking system call. Only one thread can access the Kernel at a time, so multiple threads are unable to run in parallel on multi processors. In this case, the library creates only one kernel level thread (task_struct). No matter how many user-level threads are created they all share the same kernel level thread, much like the processes running on a single core CPU.



ONE-TO-ONE MODEL:

In this case whenever a user level thread is created, the library asks the kernel to create a new kernel level thread. On the other hand, in the case of Linux, the library will use a clone system call to create a kernel level thread. Through this, it provides more concurrency than the many-to-one model by allowing another thread to run when a thread makes a blocking system call; it also allows multiple threads to run in parallel on multiprocessors.



Process/thread scheduling parameters

Linux distributions like Debian today use the Completely Fair Scheduler (CFS) as the default scheduler. Each task has a scheduling policy and a static scheduling priority associated with it. The scheduler checks the task lists maintained for each possible `sched_priority` value and runs the task at the head of the list with the highest priority. Scheduling policy determines where it will be inserted into the list of tasks with equal static priority and how it moves inside this list.

The different scheduling parameters depend upon the scheduling policy for a task in question. The Linux kernel's scheduler has several different scheduling policies that can be divided into two general categories:

- **Normal scheduling:**

  For normal scheduling policies, the `sched_priority` is always set to 0. This category consists of the following -

  - **SCHED_OTHER (aka SCHED_NORMAL)** - this is the standard Linux time-sharing scheduler policy for non-realtime tasks. Threads are chosen based on a dynamic priority determined by the **`nice`** value, which ranges from -20 (high priority) to +19 (low).
  - **SCHED_BATCH** - similar to **SCHED_OTHER**, but the scheduler assumes thread to be CPU-intensive. A small scheduling penalty is applied when it comes to wakeup behaviour so the thread is not as highly favoured in scheduling decisions. Meant for non-interactive workloads.
  - **SCHED_IDLE** - the process **`nice`** value does not affect this policy. Meant for running jobs at extremely low priority (lower even than a +19 nice value with the **SCHED_OTHER** or **SCHED_BATCH** policies).

  The CFS tries to give all runnables process a timeslice once per a period called a latency target, which is set by the **`sched_latency_ns`** parameter (in ns). Each process thus gets a slice that is **`sched_latency_ns`** / **`n_tasks`** long. For multicore systems the latency target is actually **`sched_latency_ns * (1+log2(n_CPU))`**.

  The **`sched_min_granularity_ns`** parameter is there to ensure that the timeslice is not too small that context switching would have a greater cost than the execution. It modifies the latency target by either **`sched_min_granularity_ns * (n_process)`** in single CPU, or **`sched_min_granularity_ns * (1+log2(n_CPU))`** in multiprocessor systems. Similarly, the **`wakeup_granularity_ns`** parameter ensures that processes don't go to sleep for too short a duration.

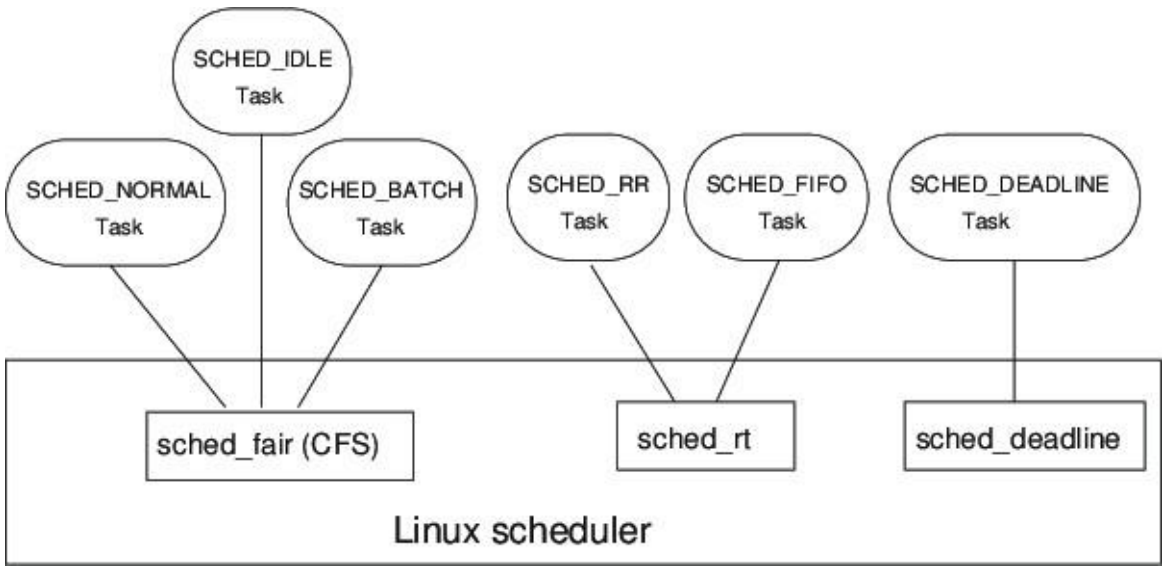
- **Real time scheduling:**

  These have `sched_priority` in range 1-99 (low to high). When one of these threads becomes runnable, they immediately preempt any running normal policy threads. A nonblocking infinite loop can block other threads from ever accessing the CPU so two parameters **`sched_rt_period_us`** and **`sched_rt_runtime_us`** are used to limit how long they can execute so they can be terminated if gone out of control. Three different real time policies are supported:

  - **SCHED_FIFO** - uses a simple first-in-first-out model where the first process (of the same priority) must either finish executing or go to sleep.
  - **SCHED_RR** - uses a round-robin scheme where each thread is allowed to run for a maximum amount of time before allowing another realtime thread to continue - ensuring they all get to execute, unlike **SCHED_FIFO**. The parameter **`sched_rr_timeslice_ms`** controls the amount of time (in ms) the thread can execute before being preempted by the next process in the round-robin.
  - **SCHED_DEADLINE** - uses a Global Earliest Deadline First strategy. This is for sporadic tasks (i.e. ones that run once per period) that must finish before a given relative deadline from its arrival time. It takes three parameters:

  i. **`sched_runtime`** - usually set to something longer than average computation time, or the worst case execution time.

  ii. **`sched_deadline`** - time to the relative deadline. iii. **`sched_period`** - time schedule period of the task



Process/thread scheduling algorithm

The CFS scheduler has a target latency, which is the minimum amount of time, idealised to an infinitely small duration, required for every runnable task to get at least one turn on the processor. If such a duration could be infinitely small, then each runnable task would have had a turn on the processor during any given timespan, however small (e.g., 10ms, 5ns, etc.). Of course, an idealised infinitely small duration must be approximated in the real world, and the default approximation is 20ms. Each runnable task then gets a 1/N slice of the target latency, where N is the number of tasks. For example, if the target latency is 20ms and there are four contending tasks, then each task gets a timeslice of 5ms. By the way, if there is only a single task during a scheduling event, this lucky task gets the entire target latency as its slice. The fair in CFS comes to the fore in the 1/N slice given to each task contending for a processor.

The 1/N slice is, indeed, a timeslice—but not a fixed one because such a slice depends on N, the number of tasks currently contending for the processor. The system changes over time. Some processes terminate and new ones are spawned; runnable processes block and blocked processes become runnable. The value of N is dynamic and so, therefore, is the 1/N timeslice computed for each runnable task contending for a processor. The traditional nice value is used to weight the 1/N slice: a low-priority nice value means that only some fraction of the 1/N slice is given to a task, whereas a high-priority nice value means that a proportionately greater fraction of the 1/N slice is given to a task. In summary, nice values do not determine the slice, but only modify the 1/N slice that represents fairness among the contending tasks.

The operating system incurs overhead whenever a context switch occurs; that is, when one process is preempted in favour of another. To keep this overhead from becoming unduly large, there is a minimum amount of time (with a typical setting of 1ms to 4ms) that any scheduled process must run before being preempted. This minimum is known as the minimum granularity. If many tasks (e.g., 20) are contending for the processor, then the minimum granularity (assume 4ms) might be more than the 1/N slice (in this case, 1ms). If the minimum granularity turns out to be larger than the 1/N slice, the system is overloaded because there are too many tasks contending for the processor—and fairness goes out the window. CFS tries to minimize context switches, given their overhead: time spent on a context switch is time unavailable for other tasks

Synchronisation tools

The POSIX Threads API, or pthreads for short, provides all the tools a program might need to handle synchronisation between threads in Linux. Using `pthread.h` we have at our disposal the following synchronisation tools - mutexes, join, condition variables, and barriers. Posix programs also can make use of semaphores defined in `semaphore.h`.

Mutex

Derived from the term *mutual exclusion*, `pthread_mutex_t` allows only one thread at a time to access a critical section. Before executing the critical section, the program calls `pthread_mutex_lock()` to lock out other threads from executing that portion of code. After finishing, the lock is released by calling `pthread_mutex_unlock()`. Another thread can now acquire lock and execute the critical section. `pthread_rwlock_t` are a specialised type of mutexes that allow for better parallelism.

Join

`pthread_join()` makes a thread wait for other threads created (using `pthread_create()`) to complete executing. This is useful when waiting to combine results from multiple threads carrying out parts of a calculation, for example.

Condition Variables

`pthread_cond_t` variables are used to make threads wait until a particular condition has been fulfilled. This is useful when a certain sequence of tasks has to be maintained e.g. data processing steps cannot start before data has been entered into a queue by another process. The function `pthread_cond_wait()` is used to wait till condition fulfilment, and upon finishing critical section execution, `pthread cond signal()` is used to signal other thread(s) to wake up.

Barriers

`pthread_barrier_t` are similar to `pthread_join`, but more general. They are used to synchronise and coordinate multiple parallely executing threads. Barriers make each thread wait until all cooperating threads reach the same point. Once all the threads have caught, they execute from there. A barrier is initialised with the number of threads that must reach the point before they are allowed to progress. `pthread_barrier_wait()` is called by a thread to indicate it is done with its part and the thread goes to sleep. Once barrier count has been reached, all threads are woken up.

Semaphores

Denoted by `sem_t`, semaphores allow multiple threads to access the critical section at once. This is useful when, say, we can allow a certain number of threads access instead of just one as we would with a mutex. Mutexes can be called binary semaphores - i.e. a semaphore that allows only one thread access at a time. Semaphores are initialised with the number of threads. When entering the critical section, `wait()` is called, which decrements the count by 1. When the count is 0, no more threads are allowed into the critical section. When finished, calling `signal()` increments the count by 1.

System generation and booting process

System generation

Installing Debian using the graphical installer is a simple process. It requires an internet connection to download the iso file and a USB flash drive for the install process.

1. Download the appropriate Debian iso from https://www.debian.org/.
2. Create the live USB by flashing the iso onto the flash drive using Rufus (https://rufus.ie).
3. Boot from live USB by changing necessary bios settings and start installer.
4. Try out the live session and see if all the hardware devices are supported and working as intended.
5. From the graphical installer, follow the steps to install Debian.

Following the installation, the system reboots and Debian has been installed on the computer.

Booting process

The system boots as follows:

1. The BIOS/UEFI bootstrap program executes. This initial bootloader locates a second bootloader in a fixed location in the disk called the boot block and loads it into memory.
2. The GRUB v2 bootloader starts execution. This program loads the kernel into memory. GRUB creates a temporary RAM file system with the necessary drivers and modules needed for the next stage and the kernel is extracted from its compressed state.
3. Kernel starts, necessary drivers are installed, the hardware is initialised.
4. Kernel mounts the root file system by switching root from the temporary system to the appropriate root file system location.
5. Finally, Linux creates the systemd process, the initial process in the system, and then starts other services

Virtual memory management

Virtual memory in Debian and other GNU/Linux distributions are called Swap. Less used RAM data is moved to disk to save RAM space when the system runs out of physical memory. The kernel creates a new virtual address space in two situations: when a process runs a new program with the exec() system call and when a new process is created by the fork() system call.

Currently Debian’s virtual memory management system makes use of paging instead of moving entire process contents at once. The Linux pageout policy gives each page an age value - frequent access gives the page a higher age. The paging algorithm decides which pages have not seen much use and then the paging mechanism swaps these pages to disk.

The paging mechanism supports paging both to dedicated swap devices and partitions and to normal files. Swapping to a file is significantly slower due to the extra overhead incurred by the file system. Blocks are allocated from the swap devices according to a bitmap of used blocks, which is maintained in physical memory at all times. The allocator uses a next-fit algorithm to try to write out pages to continuous runs of secondary storage blocks for improved performance. The allocator records that a page has been paged out to storage by using a feature of the page tables on modern processors: the page-table entry’s page-not-present bit is set, allowing the rest of the page table entry to be filled with an index identifying where the page has been written.

References

1. https://www.tutorialspoint.com/threads-vs-processes-in-linux
2. https://uomustansiriyah.edu.iq/media/lectures/5/5_2020_02_09!11_06_00_PM.pdf
3. https://www.baeldung.com/linux/process-states
4. https://opensource.com/article/19/2/fair-scheduling-linux
5. http://cs.boisestate.edu/~amit/teaching/597/scheduling.pdf
6. https://www.cs.uic.edu/~jbell/CourseNotes/OperatingSystems/5_Synchronization.html
7. https://www.linuxbaya.com/2019/04/synchronization-tools-in-operating.html?m=0
8. https://manpages.debian.org/stretch/manpages/sched.7.en.html
9. https://man7.org/linux/man-pages/man7/sched.7.html
10. https://dev.to/satorutakeuchi/the-linux-s-sysctl-parameters-about-process-scheduler-1dh5
11. https://opensource.com/article/17/2/linux-boot-and-startup
12. https://access.redhat.com/documentation/en-us/red_hat_enterprise_linux_for_real_time/7/html/reference_guide/chap-thread_synchronization
13. https://www.cs.cmu.edu/afs/cs/academic/class/15492-f07/www/pthreads.html
14. https://www.informit.com/articles/article.aspx?p=2085690&seqNum=6