Unit-2

Open Source Operating Systems: LINUX- Introduction, general overview, kernel mode and user mode, process, advanced concepts, scheduling, personalities, cloning and signals.

Unit-II

Open Source operating systems

Open Source operating systems are released under a license where the copyright holder allows others to study, change as well as distribute the software to other people. This can be done for any reason. The different open source operating system available in the market are −

Cosmos

This is an open source operating system written mostly in programming language C#. Its full form is C# Open Source Managed Operating System. Till 2016, Cosmos did not intend to be a fully fledged operating system but a system that allowed other developers to easily build their own operating systems. It also hid the inner workings of the hardware from the developers thus providing data abstraction.

FreeDOS

This was a free operating system developed for systems compatible with IBM PC computers. FreeDOS provides a complete environment to run legacy software and other embedded systems. It can booted from a floppy disk or USB flash drive as required. FreeDos is licensed under the GNU General Public license and contains free and open source software. So there is no license fees required for its distribution and changes to the system are permitted.

Genode

Genode is free as well as open source. It contains a microkernel layer and different user components. It is one of the few open source operating systems not derived from a licenced operating system such as Unix. Genode can be used as an operating system for computers, tablets etc. as required. It is also used as a base for virtualisation, interprocess communication, software development etc. as it has a small code system.

Ghost OS

This is a free, open source operating system developed for personal computers. It started as a research project and developed to contain various advanced features like graphical user interface, C library etc. The Ghost operating system features multiprocessing and multitasking and is based on the Ghost Kernel. Most of the programming in Ghost OS is done in C++.

ITS

The incompatible time-sharing system was developed by the MIT Artificial Intelligence Library. It is principally a time sharing system. There is a remote login facility which allowed guest users to informally try out the operating system and its features using ARPAnet. ITS

also gave out many new features that were unique at that time such as device independent graphics terminal, virtual devices, inter machine file system access etc.

OSv

This was an operating system released in 2013. It was mainly focused on cloud computing and was built to run on top of a virtual machine as a guest. This is the reason it doesn't include drivers for bare hardware. In the OSv operating system, everything runs in the kernel address space and there is no concept of a multi-user system.

Phantom OS

This is an operating system that is based on the concepts on persistent virtual memory and is code oriented. It was mostly developed by Russian developers. Phantom OS is not based on concepts of famous operating systems such as Unix. Its main goal is simplicity and effectiveness in process management.

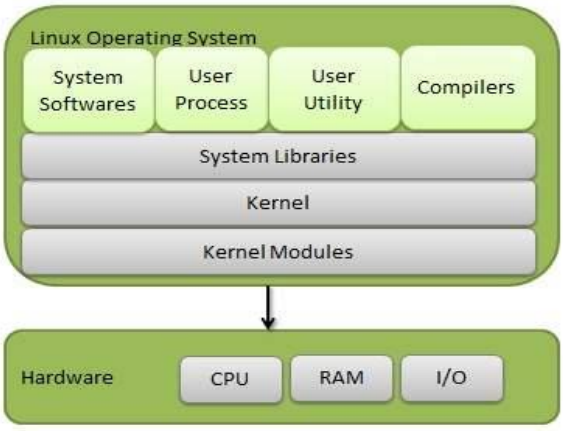
Linux Introduction

Linux is one of popular version of UNIX operating System. It is open source as its source code is freely available. It is free to use. Linux was designed considering UNIX compatibility. Its functionality list is quite similar to that of UNIX.

Components of Linux System

Linux Operating System has primarily three components

- **Kernel** − Kernel is the core part of Linux. It is responsible for all major activities of this operating system. It consists of various modules and it interacts directly with the underlying hardware. Kernel provides the required abstraction to hide low level hardware details to system or application programs.
- **System Library** − System libraries are special functions or programs using which application programs or system utilities accesses Kernel's features. These libraries implement most of the functionalities of the operating system and do not requires kernel module's code access rights.
- **System Utility** − System Utility programs are responsible to do specialized, individual level tasks.



Kernel Mode vs User Mode

Kernel component code executes in a special privileged mode called **kernel mode** with full access to all resources of the computer. This code represents a single process, executes in single address space and do not require any context switch and hence is very efficient and fast. Kernel runs each processes and provides system services to processes, provides protected access to hardware to processes.

Support code which is not required to run in kernel mode is in System Library. User programs and other system programs works in **User Mode** which has no access to system hardware and kernel code. User programs/ utilities use System libraries to access Kernel functions to get system's low level tasks.

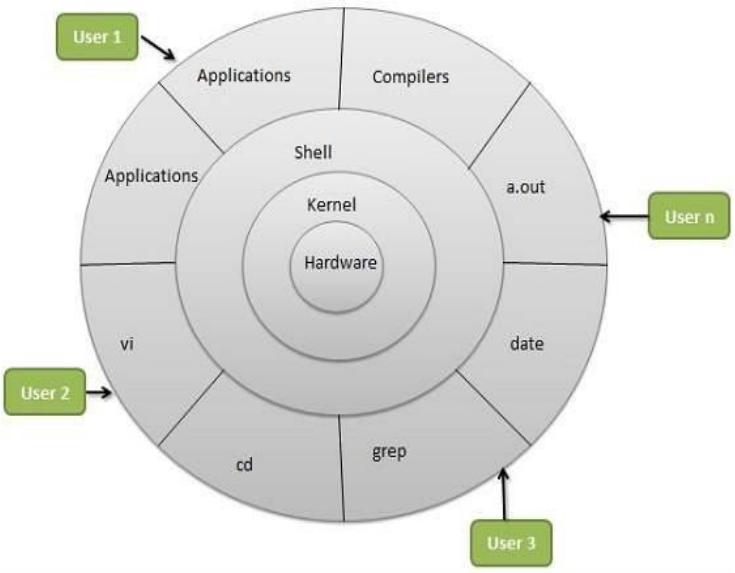
Basic Features

Following are some of the important features of Linux Operating System.

- **Portable** − Portability means software can works on different types of hardware in same way. Linux kernel and application programs supports their installation on any kind of hardware platform.
- **Open Source** − Linux source code is freely available and it is community based development project. Multiple teams work in collaboration to enhance the capability of Linux operating system and it is continuously evolving.
- **Multi-User** − Linux is a multiuser system means multiple users can access system resources like memory/ ram/ application programs at same time.
- **Multiprogramming** − Linux is a multiprogramming system means multiple applications can run at same time.
- **Hierarchical File System** − Linux provides a standard file structure in which system files/ user files are arranged.
- **Shell** − Linux provides a special interpreter program which can be used to execute commands of the operating system. It can be used to do various types of operations, call application programs. etc.
- **Security** − Linux provides user security using authentication features like password protection/ controlled access to specific files/ encryption of data.

Architecture

The following illustration shows the architecture of a Linux system −



The architecture of a Linux System consists of the following layers −

- **Hardware layer** − Hardware consists of all peripheral devices (RAM/ HDD/ CPU etc).
- **Kernel** − It is the core component of Operating System, interacts directly with hardware, provides low level services to upper layer components.
- **Shell** − An interface to kernel, hiding complexity of kernel's functions from users. The shell takes commands from the user and executes kernel's functions.
- **Utilities** − Utility programs that provide the user most of the functionalities of an operating systems.

Process in Unix/Linux

A **process** is a program in execution in memory or in other words, an instance of a program in memory. Any program executed creates a process. A program can be a command, a shell script, or any binary executable or any application. However, not all commands end up in creating process, there are some exceptions. Similar to how a file created has properties associated with it, a process also has lots of properties associated to it.

Process attributes:

A process has some properties associated to it:

PID : Process-Id. Every process created in Unix/Linux has an identification number associated to it which is called the process-id. This process id is used by the kernel to identify the process similar to how the inode number is used for file identification. The PID is unique for a process at any given point of time. However, it gets recycled.

PPID : Parent Process Id: Every process has to be created by some other process. The process which creates a process is the parent process, and the process being created is the child process. The PID of the parent process is called the parent process id(PPID).

TTY: Terminal to which the process is associated to. Every command is run from a terminal which is associated to the process. However, not all processes are associated to a terminal.There are some processes which do not belong to any terminal. These are called daemons.

UID: User Id- The user to whom the process belongs to. And the user who is the owner of the process can only kill the process(Of course, root user can kill any process). When a process tries to access files, the accessibility depends on the permissions the process owner has on those files.

File Descriptors: File descriptors related to the process: input, output and error file descriptors.

List the processes:

```
$ ps
   PID    TTY  TIME CMD
 1315012  pts/1  0:00 -ksh
 2490430  pts/1  0:00 ps
```

ps is the Unix / Linux command which lists the active processes and its status. By default, it lists the processes belonging to the current user being run from the current terminal.

The ps command output shows 4 things:
PID : The unique id of the process
TTY: The terminal from which the process or command is executed.
TIME: The amount of CPU time the process has taken
CMD: The command which is executed.

2 processes are listed in the above case:
1. -ksh : The login shell, which we are working on, is also a process which is currently running.
2. ps : The ps command which we executed to get the list also creates a process. And hence, by default, there will be atleast 2 processes when executing the ps command.

Parent & Child Process:

Every process in Unix has to be created by some other process. Hence, the ps command is also created by some other process. The 'ps' command is being run from the login shell, ksh. The ksh shell is a process running in the memory right from the moment the user logged in. So, for all the commands trigerred from the login shell, the login shell will be the parent process and the process created for the command executed will be the child process. In the same lines, the 'ksh' is the parent process for the child process 'ps'.

- The below command shows the process list along with the PPID.

```
$ ps -o pid,ppid,args
   PID    PPID COMMAND
2666744 3317840 ps -o pid,ppid,args
3317840 1       -ksh
```

The PID of the ksh is same as the PPID of the ps command which means the ksh process is the parent of the ps command. The '-o' option of the ps command allows the user to specify only the fields which he needs to display.

Init Process:

If all processes of the user are created by the login shell, who created the process for the login shell? In other words, which is the parent process of the login shell? When the Unix system boots, the first process to be created is the init process. This init process will have the PID as 1 and PPID as 0. All the other processes are created by the init process and gets branched from there on. Note in the above command, the process of the login shell has the PPID 1 which is the PID of the init process.

Exceptions to creating process:

Not all commands end up creating a process. There are some exceptions.

i) Internal commands does not create a process since they are shell built-in.

ii) Any file if sourced does not create a process since it has to be run within the shell.

Scheduling

The scheduler is the component of the kernel that selects which process to run next. The scheduler (or process scheduler, as it is sometimes called) can be viewed as the code that divides the finite resource of processor time between the runnable processes on a system. The scheduler is the basis of a multitasking operating system such as Linux. By deciding what process can run, the scheduler is responsible for best utilizing the system and giving the impression that multiple processes are simultaneously executing.

The idea behind the scheduler is simple. To best utilize processor time, assuming there are runnable processes, a process should always be running. If there are more processes than processors in a system, some processes will not always be running. These processes are waiting to run. Deciding what process runs next, given a set of runnable processes, is a fundamental decision the scheduler must make.

Multitasking operating systems come in two flavours: cooperative multitasking and preemptive multitasking. Linux, like all UNIX variants and most modern operating systems, provides preemptive multitasking. In preemptive multitasking, the scheduler decides when a process is to cease running and a new process is to resume running. The act of involuntarily suspending a running process is called preemption. The time a process runs before it is preempted is predetermined, and is called the time slice of the process. The time slice, in effect, gives each process a slice of the processor's time. Managing the time slice enables the scheduler to make global scheduling decisions for the system. It also prevents any one process from monopolizing the system. As we will see, this time slice is dynamically calculated in the Linux scheduler to provide some interesting benefits.

Conversely, in cooperative multitasking, a process does not stop running until it voluntary decides to do so. The act of a process voluntarily suspending itself is called yielding. The shortcomings of this approach are numerous: The scheduler cannot make global decisions regarding how long processes run, processes can monopolize the processor for longer than the user desires, and a hung process that never yields can potentially bring down the entire system. Thankfully, most operating systems designed in the last decade have provided preemptive multitasking, with Mac OS 9 and earlier being the most notable exceptions. Of course, Unix has been preemptively multitasked since the beginning.

During the 2.5 kernel series, the Linux kernel received a scheduler overhaul. A new scheduler, commonly called the O(1) scheduler because of its algorithmic behavior, solved the shortcomings of the previous Linux scheduler and introduced powerful new features and performance characteristics. In this section, we will discuss the fundamentals of scheduler design and how they apply to the new O(1) scheduler and its goals, design, implementation, algorithms, and related system calls.

Policy

Policy is the behavior of the scheduler that determines what runs when. A scheduler's policy often determines the overall feel of a system and is responsible for optimally utilizing processor time. Therefore, it is very important.

I/O-Bound Versus Processor-Bound Processes

Processes can be classified as either I/O-bound or processor-bound. The former is characterized as a process that spends much of its time submitting and waiting on I/O requests. Consequently, such a process is often runnable, but only for short periods, because it will eventually block waiting on more I/O (this is any type of I/O, such as keyboard activity, and not just disk I/O). Conversely, processor-bound processes spend much of their time executing code. They tend to run until they are preempted because they do not block on I/O requests very often. Because they are not I/O-driven, however, system response does not dictate that the scheduler run them often. The scheduler policy for processor-bound processes, therefore, tends to run such processes less frequently but for longer periods. Of course, these classifications are not mutually exclusive. The scheduler policy in Unix variants tends to explicitly favor I/O-bound processes.

The scheduling policy in a system must attempt to satisfy two conflicting goals: fast process response time (low latency) and high process throughput. To satisfy these requirements, schedulers often employ complex algorithms to determine the most worthwhile process to run, while not compromising fairness to other, lower priority, processes. Favoring I/O-bound processes provides improved process response time, because interactive processes are I/O-

bound. Linux, to provide good interactive response, optimizes for process response (low latency), thus favoring I/O-bound processes over processor-bound processors. As you will see, this is done in a way that does not neglect processor-bound processes.

Process Priority

A common type of scheduling algorithm is priority-based scheduling. The idea is to rank processes based on their worth and need for processor time. Processes with a higher priority will run before those with a lower priority, while processes with the same priority are scheduled round-robin (one after the next, repeating). On some systems, Linux included, processes with a higher priority also receive a longer timeslice. The runnable process with timeslice remaining and the highest priority always runs. Both the user and the system may set a processes priority to influence the scheduling behavior of the system.

Linux builds on this idea and provides dynamic priority-based scheduling. This concept begins with the initial base priority, and then enables the scheduler to increase or decrease the priority dynamically to fulfill scheduling objectives. For example, a process that is spending more time waiting on I/O than running is clearly I/O bound. Under Linux, it receives an elevated dynamic priority. As a counterexample, a process that continually uses up its entire timeslice is processor bound—it would receive a lowered dynamic priority.

The Linux kernel implements two separate priority ranges. The first is the nice value, a number from –20 to 19 with a default of zero. Larger nice values correspond to a lower priority—you are being nice to the other processes on the system. Processes with a lower nice value (higher priority) run before processes with a higher nice value (lower priority). The nice value also helps determine how long a timeslice the process receives. A process with a nice value of –20 receives the maximum timeslice, whereas a process with a nice value of 19 receives the minimum timeslice. Nice values are the standard priority range used in all Unix systems.

The second range is the real-time priority, which will be discussed later. By default, it ranges from zero to 99. All real-time processes are at a higher priority than normal processes. Linux implements real-time priorities in accordance with POSIX. Most modern Unix systems implement a similar scheme.

O(1) is an example of big-o notation. Basically, it means the scheduler can do its thing in constant time, regardless of the size of the input. A full explanation of big-o notation is in Appendix D, for the curious.

Timeslice

The timeslice is the numeric value that represents how long a task can run until it is preempted. The scheduler policy must dictate a default timeslice, which is not simple. A

timeslice that is too long will cause the system to have poor interactive performance; the system will no longer feel as if applications are being concurrently executed. A time slice that is too short will cause significant amounts of processor time to be wasted on the overhead of switching processes, as a significant percentage of the system's time will be spent switching from one process with a short time slice to the next. Furthermore, the conflicting goals of I/O-bound versus processor-bound processes again arise; I/O-bound processes do not need longer time slices, whereas processor-bound processes crave long time slices (to keep their caches hot, for example).

With this argument, it would seem that any long time slice would result in poor interactive performance. In many operating systems, this observation is taken to heart, and the default time slice is rather low—for example, 20ms. Linux, however, takes advantage of the fact that the highest priority process always runs. The Linux scheduler bumps the priority of interactive tasks, enabling them to run more frequently. Consequently, the Linux scheduler offers a relatively high default time slice. Furthermore, the Linux scheduler dynamically determines the time slice of a process based on priority. This enables higher priority, allegedly more important, processes to run longer and more often. Implementing dynamic time slices and priorities provides robust scheduling performance.

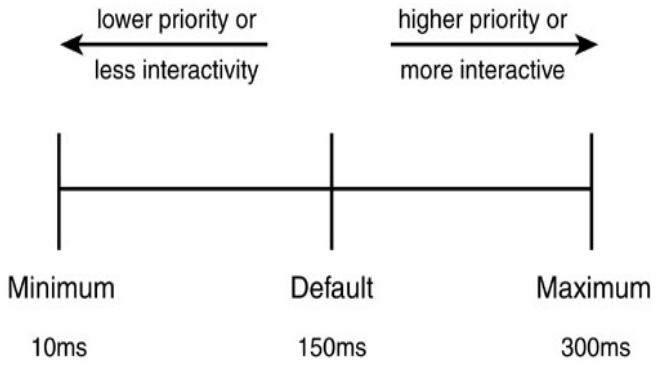


Figure 1. Process timeslice calculation.

Note that a process does not have to use all its time slice at once. For example, a process with a 100 millisecond time slice does not have to run for 100 milliseconds in one go or risk losing the remaining time slice. Instead, the process can run on five different reschedules for 20 milliseconds each. Thus, a large time slice also benefits interactive tasks—while they do not need such a large time slice all at once, it ensures they remain runnable for as long as possible.

When a process's time slice runs out, the process is considered expired. A process with no time slice is not eligible to run until all other processes have exhausted their time slice (that is, they all have zero time slice remaining). At that point, the time slices for all processes are

recalculated. The Linux scheduler employs an interesting algorithm for handling time slice exhaustion that is discussed in a later section.

Time slice is sometimes called quantum or processor slice in other systems. Linux calls it time slice.

Process Preemption

As mentioned, the Linux operating system is preemptive. When a process enters the TASK_RUNNING state, the kernel checks whether its priority is higher than the priority of the currently executing process. If it is, the scheduler is invoked to pick a new process to run (presumably the process that just became runnable). Additionally, when a process's timeslice reaches zero, it is preempted, and the scheduler is invoked to select a new process.

The Scheduling Policy in Action

Consider a system with two runnable tasks: a text editor and a video encoder. The text editor is I/O-bound because it spends nearly all its time waiting for user key presses (no matter how fast the user types, it is not that fast). Despite this, when it does receive a key press, the user expects the editor to respond immediately. Conversely, the video encoder is processor-bound. Aside from reading the raw data stream from the disk and later writing the resulting video, the encoder spends all its time applying the video codec to the raw data. It does not have any strong time constraints on when it runs—if it started running now or in half a second, the user could not tell. Of course, the sooner it finishes the better.

In this system, the scheduler gives the text editor a higher priority and larger time slice than the video encoder, because the text editor is interactive. The text editor has plenty of time slice available. Furthermore, because the text editor has a higher priority, it is capable of preempting the video encoder when needed. This ensure the text editor is capable of responding to user key presses immediately. This is to the detriment of the video encoder, but because the text editor runs only intermittently, the video encoder can monopolize the remaining time. This optimizes the performance of both applications.

Personalities

Linux supports different execution domains, or personalities, for each process. Among other things, execution domains tell Linux how to map signal numbers into signal actions. The execution domain system allows Linux to provide limited support for binaries compiled under other UNIX-like operating systems. If persona is not 0xffffffff, then personality () sets the caller's execution domain to the value specified by persona. Specifying persona as 0xffffffff provides a way of retrieving the current persona without changing it.

At any time, each process has an *effective user ID*, a *effective group ID*, and a set of *supplementary group IDs*. These IDs determine the privileges of the process. They are

collectively called the *persona* of the process, because they determine "who it is" for purposes of access control.

Your login shell starts out with a persona which consists of your user ID, your default group ID, and your supplementary group IDs (if you are in more than one group). In normal circumstances, all your other processes inherit these values.

A process also has a *real user ID* which identifies the user who created the process, and a *real group ID* which identifies that user's default group. These values do not play a role in access control, so we do not consider them part of the persona. But they are also important.

Finally, there are many operations which can only be performed by a process whose effective user ID is zero. A process with this user ID is a *privileged process*. Commonly the user name root is associated with user ID 0, but there may be other user names with this ID.

The execution domain is a 32-bit value in which the top three bytes are set aside for flags that cause the kernel to modify the behaviour of certain system calls so as to emulate historical or architectural quirks. The least significant byte is a value defining the personality the kernel should assume. The flag values are as follows:

ADDR_COMPAT_LAYOUT (since Linux 2.6.9)
With this flag set, provide legacy virtual address space layout.

ADDR_NO_RANDOMIZE (since Linux 2.6.12) With this flag set, disable address-space-layout randomization.

ADDR_LIMIT_32BIT (since Linux 2.2) Limit the address space to 32 bits.

ADDR_LIMIT_3GB (since Linux 2.4.0) With this flag set, use 0xc0000000 as the offset at which to search a virtual memory chunk on mmap(2); otherwise use 0xffffe000.

Cloning and signals

In Linux, clone() is a new, versatile system call which can be used to create a new thread of execution. Depending on the options passed, the new thread of execution can adhere to the semantics of a UNIX process, a POSIX thread, something in between, or something completely different (like a different container).

In computing, a clone is hardware or software that is designed to function in exactly the same way as another system. A specific subset of clones are remakes (or remades), which are revivals of old, obsolete, or discontinued products.

Creating Clones

Although fork() is the traditional way of creating new processes in Unix, Linux also provides the clone() system call, which lets the process being duplicated specify what resources the parent process should share with its children.

```
int clone(int flags);
```

This is not much more than a fork(); the only difference is the flags parameter. It should be set to the signal that should be sent to the parent process when the child exits (usually, SIGCHLD), bitwise OR'ed with any number of the following flags, which are defined in <sched.h>:

CLONE_VM	The two processes share their virtual memory space (including the stack).
CLONE_FS	File-system information (such as the current directory) is shared.
CLONE_FILES	Open files are shared.
CLONE_SIGHAND	The signal handlers are shared for the two processes.

When two resources are shared, both processes see those resources identically. If the CLONE_SIGHAND is specified, when one of the processes changes the signal handler for a particular signal, both processes use the new handler. When CLONE_FILES is used, not only is the set of open files shared, but the current location in each file is also shared. The return values for clone() are the same as for fork().

If a signal other than SIGCHLD is specified to be delivered to the parent on the death of a child process, the wait() family of functions will not, by default, return information on those processes. If you would like to get information on such processes, as well as on processes that use the normal SIGCHLD mechanism, the __WCLONE flag must be OR'ed with the flags parameter of the wait call. Although this behavior may seem odd, it allows greater flexibility. If wait () returned information on cloned processes, it would be more difficult to build standard thread libraries around clone(), because wait() would return information on other threads, as well as child processes.

Linux Signals

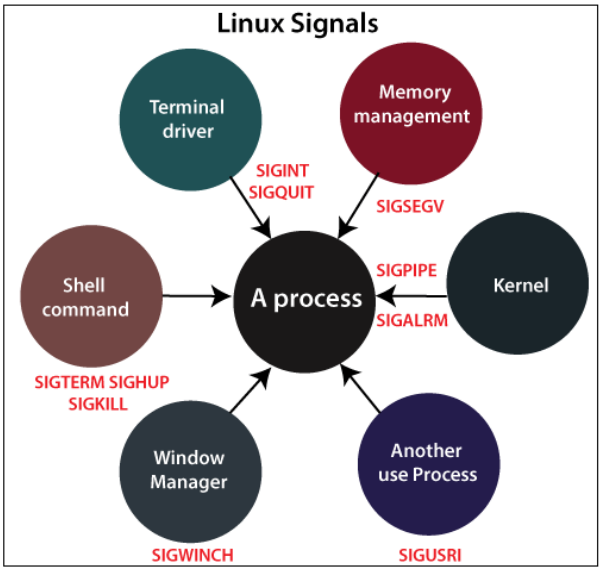
In Linux, Signals are the interrupts that are sent to the program to specify that an important event has occurred. Events can vary from user requests to invalid memory access errors. Various signals, like the interrupt signal, means that the user has asked the program to perform something which is not present in the user flow of control.

There are two kinds of signals:

1. Maskable
2. Non-Maskable

   Maskable: **-** Maskable Signals are the signals that the user can change or ignore, for example, **Ctrl +C.**

   Non-Maskable: **-** Non-Maskable Signals are the signals that the users cannot change or ignore. Non-Maskable signals mainly occur if a user is signaled for non-recoverable hardware errors.



There are various processes in different states in the Linux computer system. All these processes belong to either the operating system or the user application. A mechanism is required for the kernel and these processes to coordinate their activities. One method to perform this, for the process, to inform others if anything vital occurs. That's the reason we have signals.

Basically, the signal means a one-way notification. The kernel sent the signal to the process, to one process to itself, or another process.
Linux signals trace their origin to UNIX signals. In later Linus versions, real-time signals were added. Signal inter process is an easy and lightweight form of communication and is therefore compatible with embedded systems.

What is the Typical Lifecycle of a Signal?

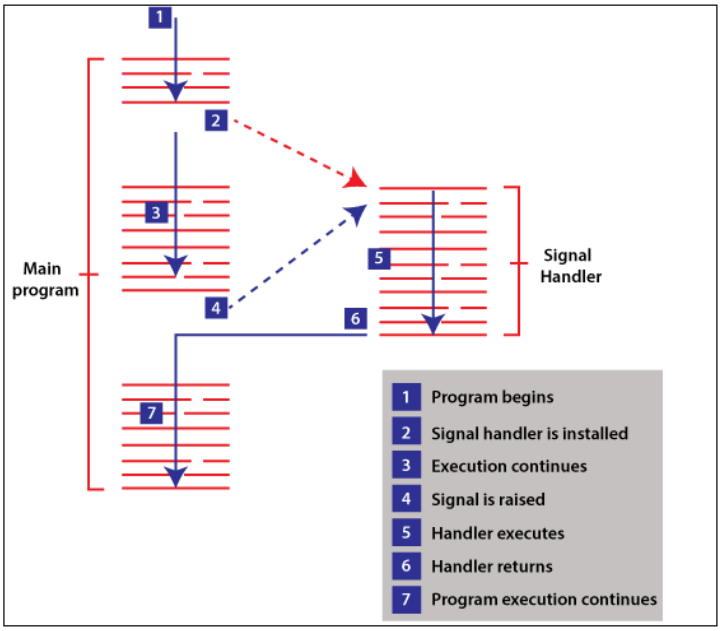
A signal goes through three stages:

1. Generation
2. Delivery
3. Processing

S.NO.	Signal Name	Meaning
1.	SIGHUP	**HUP** is a short form of **"hang up."** Locate the terminal to be controlled or hung up on the death of the control process. This signal is received when the process runs from the terminal, and that terminal goes abruptly.
2.	SIGINT	This signal is released whenever the user sends an interrupt signal **(Ctrl + C).**
3.	SIGQUIT	The SIGQUIT signal is issued when the user sends the quite signal **(Ctrl + D).**
4.	SIGILL	It is an Illegal instruction. There is some machine code that is contained in the program, and the CPU cannot understand this code.
5.	SIGTRAP	Mostly the **SIGTRAP** is used from inside debuggers and program tracers.
6.	SIGABRT	The function named **abort ()** is called by the program. It is an emergency stop.
7.	SIGBUS	SIGBUS is an attempt via which memory is accessed incorrectly. In memory access, it may cause Alignment errors.
8.	SIGFPE	The **SIGFPE** is the floating-point exception that occurred in the program.
9.	SIGKILL	When the process receives the **SIGKILL** signal, it has to quit instantly and cannot perform any of the clean-up operations.
10.	SIGUSR1	Left to programmers to do whatever they need.
11.	SIGSEGV	It was attempted to use memory that had not been allocated in the process. This was due to the end readings of arrays etc.
12.	SIGUSR2	Left to programmers to do whatever they need.
13.	SIGPIPE	**SIGPIPE** signal is used when a process generates output which is being fed to other process consume it through a pipe **("{producer | consumer"),** and then the consumer dies and this signal is then sent to the producer.

14.	SIGALRM	In this, the process may eventually request a **"wake up call"** from the operating system in the future by calling the function named **alarm ().** When that time comes, this signal is contained in the wake-up calls.
15.	SIGTERM	This process was apparently done by someone killing the program.
16.	SIGCHLD	This process previously creates one or more child processes with the **fork ()** function.
17.	SIGCONT	*(can be read in conjunction with SIGSTOP)* When a process is interrupted by sending it the **SIGSTOP** signal, we must then give it the **SIGCONT** signal to restart it.
18.	SIGSTOP	(can be read in conjunction with **SIGCONT.**) If a **SIGSTOP** signal, is sent to the process, then it is stopped by the operating system. All of it, the state is ready to restart it (by **SIGCONT**), but until then it gets another CPU cycle.
19.	SIGTSTP	**SIGSTP** is a short form for "terminal stop." Basically, the **SIGTSTOP** signal is the same as the **SIGTSTP** signal. When the user presses **Ctrl +Z** on the terminal, the **SIGTSTP** signal is sent.
20.	SIGTTIN	The only difference between **SIGTSTP** and **SIGSTOP** is that pausing is only the default action for **SIGTSTP** but is the needed action for **SIGSTOP**. This process may choose to handle **SIGTSTP** differently but is not found options about **SIGSTOP.**
21.	SIGTOU	When a background process attempts to write output to its terminal. **SIGTOU** signal is sent by the operating system. The typical response is based on the **SIGTTIN.**
22.	SIGURG	With the help of the network connection, the **SIGURG** signal is sent by the operating system whenever the "urgent" out-of- band data is sent.
23.	SIGXCPU	The **SIGXCPU** signal is sent by the operating system to the process, which crosses its CPU limit. With the shell command ("ulimit -t unlimited"), we can cancel the CPU limit before making a run, although there are more chances that something went wrong if you reach the CPU limit in make.

24.	SIGXFSZ	If there is a process that tried to create a file that is above the file format, then the operating system sends a **SIGXFSZ** signal. We are also able to cancel the limit of the file size by using the shell command ("ulimit -t unlimited") although it is more likely that something went wrong if we reached the file size limit before the run.
25.	SIGVTALRM	Both **SIGVTALR** and **SIGALRM** signals are the same, except that the **SIGALRM** signal is sent after some amount of real-time has passed while **SIGVTALRM** is sent after some time has been spent to run the process.
26.	SIGPROF	**SIGPROG** is just like **SIGVTALRM** and **SIGALRM**, whereas **SIGALRM** is sent after some amount of real-time has passed, **SIGPROG** is sent after spending some amount of time to run the process, and the system code run on behalf of the process.
27.	SIGWINCH	The **SIGWINCH** signal is used by the process when any of its windows are resized.
28.	SIGIO	(**SIGIO** is also called **SIGPOLL**). A process that can arrange for this signal to be sent is ready to do this when there is some input process or an output channel has agreed to write.
29.	SIGPWR	A power management service sent this signal to the processes in order to show that power had been changed to a short-term emergency power supply.
30.	SIGSYS	Unused



1. Generation

A signal is generated by a process via the kernel. Whichever generates the signal addresses process to a particular process. With the help of the process number, the signal is represented, and it does not contain any additional data or arguments. So, signals are lightweight. However, we can pass the additional signal with POSIX real-time signals. Functions and system calls that can produce signals contain **kill, sigqueue, raise, pthread_kill, kill,** and **tgkill.**

2. Delivery

We said that the signal remains pending until it is delivered. Basically, in a small amount of time signal is delivered to the process by the kernel. If the process has blocked the signal, the process will be pending until unblocked.

3. Processing

When the signal is delivered, it is processed in various ways. Each signal contains a default action such as terminate the process, ignore the signal, stop the process, and continue the process. For non-default behaviour, handler function might be called. Precisely which of these happens is stated through **sigaction** function.

Signals List with Meaning

The below table shows the list of the signals along with their meaning:

List of Signal

There is a simple method to list each signal that is supported by our system. We have to enter the kill -l command on the terminal, and then it will show you the list of all the supported signals:

```
javatpoint>kill -l
 1) SIGHUP       2) SIGINT       3) SIGQUIT      4) SIGILL       5) SIGTRAP
 6) SIGABRT      7) SIGBUS       8) SIGFPE       9) SIGKILL     10) SIGUSR1
11) SIGSEGV     12) SIGUSR2     13) SIGPIPE     14) SIGALRM     15) SIGTERM
16) SIGSTKFLT   17) SIGCHLD     18) SIGCONT     19) SIGSTOP     20) SIGTSTP
21) SIGTTIN     22) SIGTTOU     23) SIGURG      24) SIGXCPU     25) SIGXFSZ
26) SIGVTALRM   27) SIGPROF     28) SIGWINCH    29) SIGIO       30) SIGPWR
31) SIGSYS      34) SIGRTMIN    35) SIGRTMIN+1  36) SIGRTMIN+2  37) SIGRTMIN+3
38) SIGRTMIN+4  39) SIGRTMIN+5  40) SIGRTMIN+6  41) SIGRTMIN+7  42) SIGRTMIN+8
43) SIGRTMIN+9  44) SIGRTMIN+10 45) SIGRTMIN+11 46) SIGRTMIN+12 47) SIGRTMIN+13
48) SIGRTMIN+14 49) SIGRTMIN+15 50) SIGRTMAX-14 51) SIGRTMAX-13 52) SIGRTMAX-12
53) SIGRTMAX-11 54) SIGRTMAX-10 55) SIGRTMAX-9  56) SIGRTMAX-8  57) SIGRTMAX-7
58) SIGRTMAX-6  59) SIGRTMAX-5  60) SIGRTMAX-4  61) SIGRTMAX-3  62) SIGRTMAX-2
63) SIGRTMAX-1  64) SIGRTMAX
```

Default Actions

There is a default action that is associated with each signal. For a signal, default action means the action which a program or script performs when it receives a signal.

There are various possible default actions:

- Stop the process.
- Terminate the process
- Continue a stopped process.
- Ignore the signal
- Dump core: This default action generates a file named core which comprise the process's memory image when the process received the signal.

Sending Signals

The Number of methods is used to send signals to a script or program. For a user, the most basic method is to press the ***INTERRUPT key*** or ***CONTROL-C*** when the script execute.

When we enter ***Ctrl+C,*** the **SIGINT** signal is sent to the script, and, according to the defined default action, the script terminates.

By using the kill command, we can also deliver the signals.

We can also use another method in order to deliver the signal in which we can use the ***kill command;*** the following is the syntax of the kill command:

1. **$ kill -signal pid**

In the above syntax of the kill command, the signal means the signal's name or number, which has to be delivered, and ***pid*** means the process's ID that the signal has to be sent. For an example-

```
javatpoint>kill -1 1001
```

The above command sends the signal named hang-up or HUP to the program, which is running with ***the 1001 process ID.*** In order to send the kill signal to a similar process, we have to use the below command-

```
javatpoint>kill -9 1001
```

The above-mentioned ***process ID 1001.*** command will kill the process which is running with process **ID 1001.**

When we enter Ctrl +C or the break key on our terminal during the execution of the shell program, that program is usually terminated immediately, and our command prompt returns. It might not be desirable every time. For example, we can leave a set of temporary files that will not be cleaned.

It is so simple to trap these signals, and below is the syntax of the trap command:

1. **$ trap commands signals**

Here, the command can be any of the valid Linux command, or a user-defined function, and the signal involves any signal which we need to trap.

In shell scripts, there are two basic uses of trap:

- Clean up temporary files
- Ignore signal

Null Signal

An interesting use case of sending signals is to investigate the existence of the process. When the **kill()** system is called along with the signal argument as 0 that is a null signal, then no signal is sent, but it can just perform error checking in order to check if the process can be signaled. This means that we are able to use this procedure in order to see the process's existence. At the time, a null signal is sent, then any of the following response can occur:

1. If an error ESRCH arises, which means the target process does not exit.
2. If the call is successful, it implies that the target process is present, and the caller is allowed to send a signal to it.
3. If the error EPERM arises, then it means that the target process exists, but we do not have enough permission in order to send the signal to the process.

What Happens When a Signal Arrives?

At the time, when the signal is close to being delivered, then any of the following default actions happen, based on the signal.

1. The signal is ignored, which is, the kernel discarded it, and there is an effect on the process. (The process is unaware that the event still took place.)
2. The process is terminated, i.e., abnormal process termination, as opposed to a normal process termination which arises when a program terminates using the exit() function.
3. A core dump file is generated, and the process terminates.
4. The process's execution is resumed or suspended.
   Rather than accepting the default actions of a specific signal, a process may set the action of a signal by modifying the action which arises at the time when the signal is delivered. A program can set any of the following actions:
   1. The default action must arise. This is helpful for undoing the first modification of the signal to something other than its default.
   2. The signal is unnoticed rather than the default action to end the process.
   3. An established signal handler can be executed. The signal handler means the custom-tailored function which performs the right functions in response to signal delivery. Informing the kernel that the handler function has to be invoked if a signal arrives is called installing or establishing a signal handler. Until one of these signals has a default disposition, we cannot determine the disposition of the signal in order to terminate or dump core.

   There are two signals **SIGSTOP** and **SIGKILL,** that cannot be blocked, ignored, or caught.