Unit 3 Process Management

Process Definition

A **process** in Linux is nothing but a program in execution. It's a running instance of a program. Any command that you execute starts a process. It generally takes an input, processes it and gives us the appropriate output.

There are basically 2 types of processes

1. **Foreground processes:** Such kinds of processes are also known as **interactive processes**. These are the processes which are to be executed or initiated by the user or the programmer; they can not be initialized by system services. Such processes take input from the user and return the output. While these processes are running we cannot directly initiate a new process from the same terminal.
2. **Background processes:** Such kind of processes is also known as **non interactive processes**. These are the processes that are to be executed or initiated by the system itself or by users, though they can even be managed by users. These processes have a unique PID or process if assigned to them and we can initiate other processes within the same terminal from which they are initiated.

Process States in Linux

A process in Linux can go through different states after it's created and before it's terminated. These states are:

Running - A process in running state means that it is running or it's ready to run.

Sleeping - The process is in a sleeping state when it is waiting for a resource to be available.

Interruptible sleep - A process in Interruptible sleep will wakeup to handle signals

Uninterruptible sleep - whereas a process in Uninterruptible sleep will not.

Stopped - A process enters a stopped state when it receives a stop signal.

Zombie - Zombie state is when a process is dead but the entry for the process is still present in the table

Commands for Process Management in Linux

There are two commands available in Linux to track running processes. These two commands are **Top** and **Ps**.

1. The top Command

To track the running processes on your machine you can use the top command.

```
$ top
```

Top command displays a list of processes that are running in real-time along with their memory and CPU usage

- **PID**: Unique Process ID given to each process.
- **User**: Username of the process owner.
- **PR**: Priority given to a process while scheduling.
- **NI:** 'nice' value of a process.
- **VIRT**: Amount of virtual memory used by a process.
- **RES**: Amount of physical memory used by a process.
- **SHR**: Amount of memory shared with other processes.
- **S**: state of the process

    - ‘D’ = uninterruptible sleep
    - ‘R’ = running
    - ‘S’ = sleeping
    - ‘T’ = traced or stopped
    - ‘Z’ = zombie
- **%CPU**: Percentage of CPU used by the process.
- **%MEM**; Percentage of RAM used by the process.
- **TIME+:** Total CPU time consumed by the process.
- **Command**: Command used to activate the process.

2. ps command

ps command is short for ‘Process Status’. It displays the currently-running processes. However, unlike the top command, the output generated is not in realtime.

$ ps

PID	process ID
TTY	terminal type
TIME	total time the process has been running
CMD	name of the command that launches the process

Options of ps command ps – u ps – A

3. Stop a process

To stop a process in Linux, use the '**kill’** command. kill command sends a signal to the process.

There are different types of signals that you can send. However, the most common one is ‘kill -9’ which is ‘**SIGKILL**’.

You can list all the signals using:

The syntax for killing a process is:

```
$ kill [pid]
```

Alternatively you can also use :

```
$ kill -9 [pid]
```

This command will send a ‘SIGKILL’ signal to the process. This should be used in case the process ignores a normal kill request.

jobs - To get the list of jobs that are either running or stopped.

bg - To run all the pending and force stopped jobs in the background.

ps -ef | grep sleep - To get details of a process running in background.

Fg - To run all the pending and force stopped jobs in the foreground.

nice -n 5 sleep 100 - To run processes with priority.

Process identifiers

(PIDs) are unique values that are automatically assigned to processes on a Linux system. PIDs start from 0. The process that has the id 0 is part of the kernel and is not regarded as a normal user-mode process.
The process with the ID of the value 1 is the init process. The init process is the first process that is sprung up when a system starts. There is a maximum limit for the number of processes a system can have. Usually, this depends on the memory capacity of the system.

Process Table

The Process Table (PT) contains an entry for each process present in the system. The entry is created when the process is created by a Fork system call. Each entry contains several fields that stores all the information pertaining to a single process. The maximum number of entries in PT (which is maximum number of processes allowed to exist at a single point of time in eXpOS) is MAX_PROC_NUM. In the current version of eXpOS, MAX_PROC_NUM = 16.

Offset	0	1	2	3	4	6	7	8	9	10	11	12	13	14	15
Field Name	TICK	PID	PPID	USERID	STATE	SWAP FLAG	INODE INDEX	INPUT BUFFER	MODE FLAG	USER AREA SWAP STATUS	USER AREA PAGE NUMBER	KERNEL STACK POINTER (KPTR)	USER STACK POINTER (UPTR)	PTBR	PTLR

- **TICK** (1 word)- keeps track of how long the process was in memory/ swapped state. It has an initial value of 0 and is updated whenever the scheduler is called. TICK is reset to 0 when a process is swapped out or in.
- **PID** (1 word) - process descriptor, a number that is unique to each process. This field is set by Fork System Call. In the present version of eXpOS, the pid is set to the index of the entry in the process table.
- **PPID** (1 word) - process descriptor of the parent process. This field is set by Fork System Call. PPID of a process is set to -1 when it's parent process exits. A process whose parent has exited is called an Orphan Process.
- **USERID** (1 word) - Userid of the currently logged in user. This field is set by Fork System Call.
- **STATE** (2 words) - a two tuple that describes the current state of the process. The details of the states are explained below.
- **SWAP FLAG** (1 word) - Indicates if the process is swapped (1) or not (0). The process is said to be swapped if any of its user stack pages or its kernel stack page is swapped out.
- **INODE INDEX** (1 word)- Pointer to the Inode entry of the executable file, which was loaded into the process's address space.
- **INPUT BUFFER** (1 word) - Buffer used to store the input read from the terminal. Whenever a word is read from the terminal, Terminal Interrupt Handler will store the word into this buffer.
- **MODE FLAG** (1 word) - Used to store the system call number if the process is executing inside a system call. It is set to -1 when the process is executing the exception handler. The value is set to 0 otherwise.
- **USER AREA SWAP STATUS** (1 word) - Indicates whether the user area of the process has been swapped (1) or not (0).
- **USER AREA PAGE NUMBER** (1 word) - Page number allocated for the user area of the process.

- **KERNEL STACK POINTER** (1 word) - Pointer to the top of the kernel stack of the process. The **offset of this address within the user area** is stored in this field.
- **USER STACK POINTER** (1 word) - Logical address of the top of the user stack of the process. This is used when the process is running in kernel mode and the machine's stack pointer is pointing to the top of the kernel stack.
- **PTBR** (1 word) - pointer to PER-PROCESS PAGE TABLE.
- **PTLR** (1 word) - PAGE TABLE LENGTH REGISTER of the process.

Viewing process - to Check Running Processes in Linux Using ps, top, htop, and atop Commands.

UNDERSTANDING PROCESS TYPES

There are different types of processes in a Linux system. These types include user processes, daemon processes, and kernel processes.

User Processes

Most processes in the system are user processes. A user process is one that is initiated by a regular user account and runs in user space. Unless it is run in a way that gives the process special permissions, an ordinary user process has no special access to the processor or to files on the system that don't belong to the user who launched the process.

Daemon Process

Daemons are a special type of background process that starts when the system boots and keeps running forever, till eternity — they never die. Daemons can be controlled by a user through the init process

Although daemon processes are typically managed as services by the root user, daemon processes often run as non-root users by a user account that is dedicated to the service. By running daemons under different user accounts, a system is better protected in the event of an attack. For example, if an attacker were to take over the httpd daemon (web server), which runs as the Apache user, it would give the attacker special access to files owned by other users (including root) or other daemon processes.

Systems often start daemons at boot time and have them run continuously until the system is shut down. Daemons can also be started or stopped on demand, set to run at particular system run levels, and, in some cases, signalled to reload configuration information on the fly.

Kernel Processes

Kernel processes execute only in kernel space. They are similar to daemon processes. The primary difference is that kernel processes have full access to kernel data structures, which makes them more powerful than daemon processes that run in user space.

Kernel processes also are not as flexible as daemon processes. You can change the behavior of a daemon process by changing configuration files and reloading the service. Changing kernel processes, however, may require recompiling the kernel.

Process Scheduling in Linux

Process scheduling is one of the most important aspects or roles of any operating system.

- Process Scheduling is an important activity performed by the *process manager* of the respective operating system.
- **Scheduling in Linux deals with the removal of the current process from the CPU and selecting another process for execution**.

In LINUX a Process Scheduler deals with process scheduling.

- Process Scheduler chooses a process to be executed.
- Process scheduler also decides for how long the chosen process is to be executed.

Hence, Scheduling Algorithms is nothing but a kind of strategy that helps the process scheduler in deciding the process to be executed.

Process Scheduling Types
Realtime Process
Real-time processes are processes that cannot be delayed in any situation. Real-time processes are referred to as urgent processes.

There are mainly two types of real-time processes in LINUX namely:

- SCHED_FIFO
- SCHED_RR.

A real-time process will try to seize all the other working processes having lesser priority.

For example, A *migration process* that is responsible for the distribution of the processes across the CPU is a real-time process. Let us learn about different scheduling policies used to deal with real-time processes briefly.

SCHED_FIFO
FIFO in SCHED_FIFO means First In First Out. Hence, the SCHED_FIFO policy schedules the processes according to the arrival time of the process.

SCHED_RR
RR in SCHED_RR means **Round Robin**. The SCHED_RR policy schedules the processes by giving them a fixed amount of time for execution. **This fixed time is known as time quantum**

Normal Process

Normal Processes are the opposite of real-time processes. Normal processes will execute or stop according to the time assigned by the process scheduler. Hence, a normal process can suffer some delay if the CPU is busy executing other high-priority processes. Let us learn about different scheduling policies used to deal with the normal processes in detail.

Normal (SCHED_NORMAL or SCHED_OTHER)
SCHED_NORMAL / SCHED_OTHER is the default or standard scheduling policy used in the LINUX operating system. A time-sharing mechanism is used in the normal policy. A time-sharing mechanism means assigning some specific amount of time to a process for its execution. **Normal policy deals with all the threads of processes that do not need any real-time mechanism**.

Batch (SCHED_BATCH)

As the name suggests, the SCHED_BATCH policy is used for executing a batch of processes. This policy is somewhat similar to the Normal policy. SCHED_BATCH policy deals with the non-interactive processes that are useful in optimizing the CPU throughput time. SCHED_BATCH scheduling policy is used for a group of processes having priority: 0.

Idle (SCHED_IDLE)

SCHED_IDLE policy deals with the processes having extremely Low Priority. Low-priority tasks are the tasks that are executed when there are absolutely no tasks to be executed. **SHED_IDLE policy is designed for the lowest priority tasks of the operating systems**.

What is an init process?
Now, what on earth is an init process you may ask, well it is a parent of all the processes that are happening in the system. The init process is the very first program that is executed when the system starts up and it is started by the kernel itself. Since the init process is the holy mother of all processes, it manages all the processes on the system, and it does not have a parent.
Linux Child and parent processes
Before we proceed further, let us get our terminology straight, let us look at what the terms child process and parent process mean.
Parent process in Linux
These are the processes that create other processes.
Child process in Linux
These are the processes that are created by other processes.
How are processes created in Linux?

In Linux, a process is normally created when an application or program makes an exact copy of itself in the memory. The child process will have the exact environment ad that of the parent process. The only difference between them is the ID number.

In Linux, there are 2 ways of creating a new process, let us look at them:

1. system()

The first method of creating a process in Linux is by using the function system(). This on the easiest of the 2 methods, however, it has security risks and is inefficient.

2. fork() / exec()

The second method of creating a process in Linux is by using either the fork() function or the exec() function. This method is slightly advanced yet is more flexible. It also offers security and speed.

Difference between zombie and orphan process
Now that we have discussed what an orphan and a zombie process are and how they are both formed, we will discuss their key differences below.

- **Process status:** A zombie process refers to a process that has completed its execution and waiting for its parent to collect its exit status, whereas an orphan process is still in its execution phase even after its parent terminates.
- **Parent status:** The init process adopts an orphan process, which becomes its new parent, and the zombie process continues waiting for its original parent.
- **System impact:** As zombie processes are not in the running phase, they do not use up many system resources, whereas an orphan process continues to consume system resources as they are still running.
- **Process handling:** The parent process handles a zombie process, whereas the init process handles the orphan process after its parent is terminated.

fork()
The Fork system call is used for creating a new process in Linux, and Unix systems, which is called the ***child process***, which runs concurrently with the process that makes the fork() call (parent process).

After a new child process is created, both processes will execute the next instruction following the fork() system call.
The child process uses the same pc(program counter), same CPU registers, and same open files which use in the parent process. It takes no parameters and returns an integer value.
Below are different values returned by fork().

- ***Negative Value***: The creation of a child process was unsuccessful.
- ***Zero***: Returned to the newly created child process.
- ***Positive value***: Returned to parent or caller. The value contains the process ID of the newly created child process.

Example 1:
```
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <unistd.h>
void forkexample()
{
   pid_t p;
   p = fork();
   if(p<0)
   {
     perror("fork fail");
     exit(1);
   }
   // child process because return value zero
   else if ( p == 0)
      printf("Hello from Child!\n");

   // parent process because return value non-zero.
   else
      printf("Hello from Parent!\n");
}
int main()
{
   forkexample();
   return 0;
}
```
Output
```
Hello from Parent!
Hello from Child!
```

Example 2:
```
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
int main()
{
   pid_t p = fork();
   if(p<0){
     perror("fork fail");
     exit(1);
   }
   printf("Hello world!, process_id(pid) = %d \n",getpid());
   return 0;
}
```

Output

```
Hello world!, process_id(pid) = 31
Hello world!, process_id(pid) = 32
```

Example 3: Calculate the number of times hello is printed.

```
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
int main()
{
   fork();
   fork();
   fork();
   printf("hello\n");
   return 0;
}
```

Output

```
hello

hello

hello

hello

hello

hello

hello

hello
```

Explanation

The number of times 'hello' is printed is equal to the number of processes created. Total Number of Processes = 2^n, where n is the number of fork system calls. So here n = 3, 2^3 = 8 Let us put some label names for the three lines:

```
fork ();   // Line 1

fork ();   // Line 2

fork ();   // Line 3

       L1       // There will be 1 child process

    /     \     // created by line 1.

  L2      L2    // There will be 2 child processes

   /  \      /  \   //  created by line 2

L3  L3  L3  L3  // There will be 4 child processes

                // created by line 3
```

1. fork() **:**

Fork() is system call which is used to create new process. New process created by fork() system call is called child process and process that invoked fork() system call is called parent process. Code of child process is same as code of its parent process. Once child process is created, both parent and child processes start their execution from next statement after fork() and both processes get executed simultaneously.

2. vfork(): :
Vfork() is also system call which is used to create new process. New process created by vfork() system call is called child process and process that invoked vfork() system call is called parent process. Code of child process is same as code of its parent process. Child process suspends execution of parent process until child process completes its execution as both processes share the same address space.

Difference between fork() and vfork() :

S.No.	FORK()	VFORK()
1.	In fork() system call, child and parent process have separate memory space.	While in vfork() system call, child and parent process share same address space.
2.	The child process and parent process gets executed simultaneously.	Once child process is executed then parent process starts its execution.
3.	The fork() system call uses copy-on-write as an alternative.	While vfork() system call does not use copy-on-write.
4.	Child process does not suspend parent process execution in fork() system call.	Child process suspends parent process execution in vfork() system call.
5.	Page of one process is not affected by page of other process.	Page of one process is affected by page of other process.
6.	fork() system call is more used.	vfork() system call is less used.
7.	There is wastage of address space.	There is no wastage of address space.
8.	If child process alters page in address space, it is invisible to parent process.	If child process alters page in address space, it is visible to parent process.

wait command
wait is an inbuilt command in the Linux shell. It waits for the process to change its state i.e. it waits for any running process to complete and returns the exit status.
Syntax:
wait [ID]

Here, ID is a **PID** (Process Identifier) which is unique for each running process. To find the process ID of a process you can use the command:
pidof [process_name]

Approach:
- Creating a simple process.
- Using a special variable**($!)** to find the PID(process ID) for that particular process.
- Print the **process ID.**
- Using **wait** command with process ID as an argument to wait until the process finishes.
- After the process is finished printing process ID with its **exit status**.

Note: Exist status 0 indicates that the process is executed successfully without any issue. Any value other than 0 indicates that the process has not been completed successfully.

```
#!/bin/bash
# creating simple process that will create file and write into it
cat > abc.txt <<< "Something is going to save into this file" &
# this will store the process id of the running process
# $! is a special variable in bash
# that will hold the PID of the last active process i.e. creating a file.
pid=$!
# print process is running with its PID
echo "Process with PID $pid is running"
# Waiting until process is Completed
wait $pid
# print process id with its Exit status
# $? is special variable that holds the return value of the recently executed command.
echo "Process with PID $pid has finished with Exit status: $?"
```

Output

```
Process with PID 1234 is running
Process with PID 1234 has finished with Exit status:0
```

The **waitpid**() system call suspends execution of the current process until a child specified by *pid* argument has changed state. By default, **waitpid**() waits only for terminated children, but this behaviour is modifiable via the *options* argument.

exec command

exec command in Linux is used to execute a command from the bash itself. This command does not create a new process it just replaces the bash with the command to be executed. If the exec command is successful, it does not return to the calling process.

Syntax:

exec [-cl] [-a name] [command [arguments]] [redirection ...]

Options:

c: It is used to execute the command with empty environment.

a name: Used to pass a name as the zeroth argument of the command.

l: Used to pass dash as the zeroth argument of the command.

Note: *exec* command does not create a new process. When we run the exec command from the terminal, the ongoing terminal process is replaced by the command that is provided as the argument for the exec command.

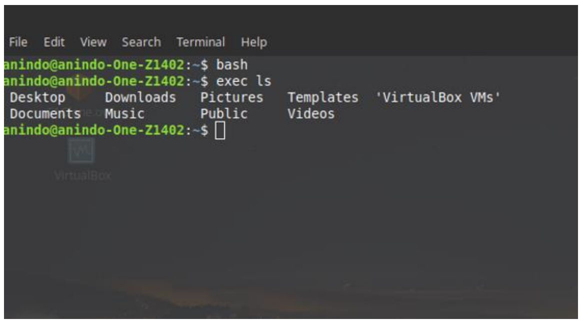
The exec command can be used in two modes:

Exec with a command as an argument:

In the first mode, the exec tries to execute it as a command passing the remaining arguments, if any, to that command and managing the redirections, if any.

Example 1:

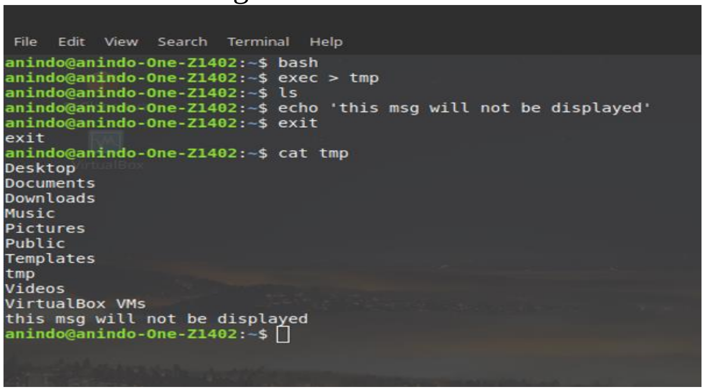
Exec ls



```
anindo@anindo-One-Z1402:~$ bash
anindo@anindo-One-Z1402:~$ exec ls
 Desktop     Downloads   Pictures   Templates  'VirtualBox VMs'
 Documents   Music       Public     Videos
anindo@anindo-One-Z1402:~$ 
```

Exec without a command:

If no command is supplied, the redirections can be used to modify the current shell environment. This is useful as it allows us to change the file descriptors of the shell as per our desire. The process continues even after the exec command unlike the previous case but now the standard input, output, and error are modified according to the redirections.



```
anindo@anindo-One-Z1402:~$ bash
anindo@anindo-One-Z1402:~$ exec > tmp
anindo@anindo-One-Z1402:~$ ls
anindo@anindo-One-Z1402:~$ echo 'this msg will not be displayed'
anindo@anindo-One-Z1402:~$ exit
exit
anindo@anindo-One-Z1402:~$ cat tmp
Desktop
Documents
Downloads
Music
Pictures
Public
Templates
tmp
Videos
VirtualBox VMs
this msg will not be displayed
anindo@anindo-One-Z1402:~$ 
```

Here the exec command changes the standard out of the shell to the tmp file and so all the commands executed after the exec command write their results in that file. This is one of the most common ways of using exec without any commands.

Signal Functions
A signal is a message or notification issued to your program by the operating system or another application (or one of its threads). Each signal is assigned a number between 1 and 31. Signals are devoid of argument, and most of the time, their names are self-explanatory. For instance, signal number 9 or SIGKILL notifies the program that it is being attempted to be killed.

List of Signals

There is a simple method for compiling a list of all the signals your system supports. Simply type the **kill -l** command to see all the allowed signals.

```
kill -l
```

Types of Signals

- **SIGHUP-** This signal indicates that the controlling terminal has been killed. HUP is an abbreviation meaning “hang up.” Locate the terminal to be controlled or hang up on the control process’s demise. This signal is obtained when the process is performed from the terminal and that terminal abruptly terminates.
- **SIGINT-** This is the signal generated when a user presses Ctrl + C from the keyboard.
- **SIGQUIT-** This is the signal generated when a user presses Ctrl + D from the keyboard.
- **SIGILL-** Signal for illegal instruction. This is an exception signal provided by the operating system to your application when it detects unlawful instruction within your program. For example, if some code is not understandable by your machine or if your program’s executable file is corrupted. Another possibility is that your program loads a corrupted dynamic library.
- **SIGABRT-** Abort signal indicates that you used the abort() API within your program. It is used to end a program. abort() generates the SIGABRT signal, which terminates your program (unless handled by your custom handler).
- **SIGFPE-** Exception for floating point numbers. Another exception signal is generated by the operating system when your program causes an exception.
- **SIGPIPE-** Broken pipe. When there is nothing to read on the other end, write to the pipe.
- **SIGSEGV-** This is also an exception signal. When a program tries to access memory that does not belong to it, the operating system gives that application this signal.

- **SIGALRM-** Alarm Signal sent through the alarm() system function to your program. The alarm() system call essentially acts as a timer that allows you to receive SIGALRM after a set amount of time. Although there are other timer APIs that are more accurate, this can be useful.
- **SIGTERM-** This signal instructs your program to quit. While SIGKILL is an abnormal termination signal, think of this as a signal to cleanly shut down.
- **SIGCHLD-** Informs you that a child's process of your program has ended or stopped. This is useful if you want to synchronize your process with one that has children.
- **SIGUSR1 and SIGUSR2-** SIGUSR1 and SIGUSR2 are two undefined signals that are provided for your consideration. These signals can be used to communicate with or synchronize your software with another program.

Signal Handler

You can register your own signal handler using one of the various interfaces.

1. signal()

The oldest one is this one. It takes two arguments: a reference to a signal handler code and a signal number (one of those SIGsomethings). A single integer input representing a sent signal number is taken by the signal handler function, which returns void. In this manner, you can apply the same signal handler function to numerous signals.

Syntax: signal(SIGINT, sig_handler);

Signal() allows you to specify the default signal handler that will be utilized for a specific signal. You can also instruct the system to disregard a certain signal. Choose SIG IGN as the signal handler if you want to ignore the signal. Set SIG DFL as the signal handler to restore the default signal handler.

Even when it appears to be everything you would need, it is preferable to stay away from employing signal (). This system call has a problem with portability. In other words, it acts differently under various operating systems. There is a more recent system call that performs all the functions of signal() while additionally providing a little bit more details about the signal itself, its origin, etc.

2. sigaction()

Another system call that modifies the signal handler is sigaction().

Syntax: *int sigaction(int signum, const struct sigaction *act, struct sigaction *oldact);*

A signal number is specified in its first argument. Pointers to the sigaction structure are provided as the second and third arguments. This structure describes how the given signal should be handled by the process.

Signal as interrupt

- Signals disrupt your program in addition to being instructive.
- For example, one of the threads in your application must briefly switch to signal handler mode in order to process a signal.
- As of the Linux kernel version 2.6, it should be noted that most signals only interrupt one thread, as opposed to the previous practice of interrupting the entire application.
- Additionally, a signal handler itself may be halted by a different signal.

Signal masks

Each signal has one of three possible states:

- For the signal, we might have our own signal handler.
- The default handler may be used to handle a signal. Each signal has a default handler job that it performs. For instance, your application will be terminated by the SIGINT default handler.
- The signal could be overlooked. Signal blocking is another name for ignoring a signal.

It is frequently simpler to handle a "signal mask" when manipulating signals and controlling signal setup. Each bit in a bit-mask has a matching signal. Since there are 32 (really 31, since 0 doesn't count) different signals, we can store information about 32 signals in a single 32-bit integer (unsigned int).

Signals that report exceptions

Signals can also be used to suggest that something unpleasant has transpired. For instance, the operating system sends a SIGSEGV signal to your application when your software creates a segmentation failure.

Unreliable Signals is when there is interruption by signal, system calls could not restart automatically and signal handler does not remain installed, while reliable signal, signal handler remains installed upon calling and condition for re-installation does not occur.

Interrupted system call

Interruption of a system call by a signal handler occurs only in the case of various blocking system calls, and happens when the system call is interrupted by a signal handler that was explicitly established by the programmer.

kill command

kill command in Linux (located in /bin/kill), is a built-in command which is used to terminate processes manually. *kill* command sends a signal to a process that terminates the process. If the user doesn't specify any signal which is to be sent along with the kill command, then a default *TERM* signal is sent that terminates the process.

Basic Syntax of `kill` command in Linux

The basic syntax of the `kill` command is as follows:
Syntax:
kill [signal] PID

- **PID =** The `kill` command requires the process ID (PID) of the process we want to terminate.
- **[signal] =** We have to specify the signal and if we don't specify the signal, the default signal **`TERM`** is sent to terminate the process

Some Common Signals in `kill` command in Linux
The table below shows some common signals and their corresponding numbers.

Signal Name	Signal Number	Description
SIGHUP	1	It hangup detected on controlling terminals or death of controlling process.
SIGINT	2	It interrupts from keyboard.
SIGKILL	9	It kills signal.
SIGTERM	15	It terminates signal.

1. `kill -l `
To display all the available signals, you can use the below command option:

2. `kill PID`
This option specifies the process ID of the process to be killed.

3. `kill -s`
This option specifies the signal to be sent to the process.

raise()

The raise() function sends the signal *sig* to the running program. If compiled with SYSIFCOPT(*ASYNCSIGNAL) on the compilation command, this function uses asynchronous signals. The asynchronous version of this function throws a signal to the process or thread.
Return Value
The raise() function returns 0 if successful, nonzero if unsuccessful.

alarm()

The *alarm()* function causes the system to generate a SIGALRM signal for the process after the number of real-time seconds specified by *seconds* have elapsed. Processor scheduling delays may prevent the process from handling the signal as soon as it is generated.

If *seconds* is 0, a pending alarm request, if any, is cancelled.

Alarm requests are not stacked; only one SIGALRM generation can be scheduled in this manner; if the SIGALRM signal has not yet been generated, the call will result in rescheduling the time at which the SIGALRM signal will be generated.

Interactions between *alarm()* and any of *setitimer()*, *ualarm()* or *usleep()* are unspecified.
RETURN VALUE
If there is a previous *alarm()* request with time remaining, *alarm()* returns a non-zero value that is the number of seconds until the previous request would have generated a SIGALRM signal. Otherwise, *alarm()* returns 0.

pause()

The *pause*() function shall suspend the calling thread until delivery of a signal whose action is either to execute a signal-catching function or to terminate the process.

If the action is to terminate the process, *pause*() shall not return.

If the action is to execute a signal-catching function, *pause*()shall return after the signal-catching function returns.

abort()

The **abort**() function first unblocks the **SIGABRT** signal, and then raises that signal for the calling process (as though raise(3) was called). This results in the abnormal termination of the process unless the **SIGABRT** signal is caught and the signal handler does not return.

If the **SIGABRT** signal is ignored, or caught by a handler that returns, the **abort**() function will still terminate the process. It does this by restoring the default disposition for **SIGABRT** and then raising the signal for a second time.
RETURN VALUE The **abort**() function never returns.

system()
The system() function has two different behaviors. These behaviors are designated as ANSI system() and POSIX system(). The ANSI system() behavior is based on the ISO C standard definition of the function, whereas the POSIX system() behavior is based on the POSIX standard definition. The ANSI system() behavior is used when a) running POSIX(OFF) or b) when running POSIX(ON) and environment variable __POSIX_SYSTEM is set to NO. Otherwise the POSIX system() behavior is used.

sleep()
The **sleep** command suspends the calling process of the next command for a specified amount of time. This property is useful when the following command's execution depends on the successful completion of a previous command.

sleep [number]

sleep(5)
In the example above, after **sleep 5** was executed, the second command prompt appeared with a 5-second delay.

Set up an Alarm
Use **sleep** to tell the system to play an mp3 file after a certain amount of time. The example uses mplayer:
sleep 7h 30m && mplayer alarm.mp3