Title : Study and implementation of signal system call.
Aim : To demonstrate signal system call.

Objective: This lab describes what are the signals, for what purpose they are used and their types.

- **Introduction**

Perhaps any engineer developing for Linux encounters this problem. What's is the right way to terminate the program? What are the ways to receive notifications from operating system about events that occur?
Traditional Unix systems have the answers ready. The answer to these questions is signals.

- **What are signals?**

Signal is a notification, a message sent by either operating system or some application to your program (or one of its threads).
Each signal identified by a number, from 1 to 31. Signals don't carry any argument and their names are mostly self-explanatory. For instance, SIGKILL or signal number 9 tells the program that someone tries to kill it.

- **Signal as interrupt**

In addition to informative nature of signals, they also interrupt your program. I.e to handle a signal, one of the threads in your program, stops its execution and temporarily switches to signal handler. Note that as in version 2.6 of Linux kernel, most of the signals interrupt only one thread and not the entire application as it used to be once. Moreover, signal handler itself can be interrupted by some other signal.

- **Signal masks**

Each one of signals can be in one of three states:

✓ We may have our own signal handler for the signal.
✓ Signal may be handled by the default handler. Every signal has its default handler function. For instance, SIGINT default handler will terminate your application.
✓ Signal may be ignored. Ignoring signal sometimes referred to as blocking signal.

When manipulating signals and managing signal configuration, it is often easier to manage a so called signal mask. It is a bit-mask, where each bit has a corresponding signal. There are 32 (actually 31, 0 doesn't count) different signals, thus we can use single 32-bit integer (unsigned int) to keep information about 32 signals.
The C library assigns default signal handlers. This means that even if you leave signals untouched, your program will process signals and will respond to them according to default behavior.

- **What signals are good for?**

As their name suggests, signals are used to signal something. There are various signal types, and each one denotes a different meaning. Each signal's designation is determined by its semantics. In other words, you might wish to select what action should be connected to each of the signals. You might determine that your application would print anything or draw something on the screen in response to a certain signal. Most of the time, it is your choice. There is, however, a standard norm for what each and every signal should do. This accepted practice states that SIGINT should force your program to end. It is in your best interest to maintain this as the SIGINT signal's default response. Usability is an issue. Nobody desires a non-interruptible program.

- **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.

- **Other uses of signals**

There are various uses for signals.

✓ Debuggers, for example, depend on signals to receive events regarding programs that are being debugged.

✓ One of the so-called IPC – Inter-Process Communication – techniques are called signals. IPC once allowed processes to communicate with one another, as the acronym suggests.

✓ Another frequent application is when a user wants our program to restart itself rather than the end. In this scenario, the user can use software called kill to send a signal to our program from the terminal. This program might be one you're already acquainted with. Previously, it killed processes. In actuality, it does convey a signal. Each signal is identified by a unique number. It can transmit any signal, but by default, it sends signal 15 or SIGTERM.

Types of signals

- **SIGHUP**
  This signal indicates that someone has killed the controlling terminal. For instance, lets say our program runs in xterm or in gnome-terminal. When someone kills the terminal program, without killing applications running inside of terminal window, operating system sends SIGHUP to the program. Default handler for this signal will terminate your program. *Thanks to Mark Pettit for the tip.*
- **SIGINT**
  This is the signal that being sent to your application when it is running in a foreground in a terminal and someone presses CTRL-C. Default handler of this signal will quietly terminate your program.
- **SIGQUIT**
  Again, according to documentation, this signal means "Quit from keyboard". In reality I couldn't find who sends this signal. I.e. you can only send it explicitly.
- **SIGILL**
  Illegal instruction signal. This is a exception signal, sent to your application by the operating system when it encounters an illegal instruction inside of your program. Something like this may happen when executable file of your program has been corrupted. Another option is when your program loads dynamic library that has been corrupted. Consider this as an exception of a kind, but the one that is very unlikely to happen.
- **SIGABRT**
  Abort signal means you used used *abort()* API inside of your program. It is yet another method to terminate your program. *abort()* issues SIGABRT signal which in its term terminates your program (unless handled by your custom handler). It is up to you to decide whether you want to use *abort()* or not.
- **SIGFPE**
  Floating point exception. This is another exception signal, issued by operating system when your application caused an exception.
- **SIGSEGV**
  This is an exception signal as well. Operating system sends a program this signal when it tries to access memory that does not belong to it.
- **SIGPIPE**
  Broken pipe. As documentation states, this signal sent to your program when you try to write into pipe (another IPC) with no readers on the other side.
- **SIGALRM**
  Alarm signal. Sent to your program using *alarm()* system call. The *alarm()* system call is basically a timer that allows you to receive SIGALRM in preconfigured number of seconds. This can be handy, although there are more accurate timer API out there.
- **SIGTERM**
  This signal tells your program to terminate itself. Consider this as a signal to cleanly shut down while SIGKILL is an abnormal termination signal.
- **SIGCHLD**
  Tells you that a child process of your program has stopped or terminated. This is handy when you wish to synchronize your process with a process with its child.
- **SIGUSR1** and **SIGUSR2**
  Finally, SIGUSR1 and SIGUSR2 are two signals that have no predefined meaning and are left for

your consideration. You may use these signals to synchronise your program with some other program or to communicate with it.

- **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.

Usage of signal()

- For the signal with a number signum, the signal() system function installs a new signal handler. The signal handler is set to sighandler, which can be either SIG_IGN or SIG_DFL or a user-specified function.
- The following occurs when a signal with the number signum arrives. The signal is disregarded if the relevant handler is set to SIG_IGN. The signal's default action takes place if the handler is set to SIG_DFL. Finally, if the handler is a function called sighandler, sighandler is executed after the handler has been reset to SIG_DFL or the signal has been blocked, depending on the implementation.
- "Catching the signal" refers to using a signal handler function for a signal. SIGKILL and SIGSTOP signals cannot be intercepted or disregarded.
- The signal() function either returns SIG ERR on failure or the previous value of the signal handler.

Example 1: Illustrating the hangup of the signal using the below command
Syntax:
kill -signal pid

Script:
#!/bin/bash

```
kill -1  500
```

Output:

Explanation:
The signal is the name or number of the signal that has to be given in the kill command syntax above, and pid is the ID of the process to which the signal needs to be sent. The program executing with the 500 process ID receives the signal called hang-up, or HUP, from the aforementioned command. If no such process is running with pid 500 then a message will be displayed as shown in the above output.
Example 2: Illustrating the killing of signal using the below command
Script:

```
#!/bin/bash

kill -9  500
```

Output:

Explanation:
Use the above command to send the kill signal to a similar process. The process running under process ID 500 will be terminated by the command already mentioned. If no such process is running with pid 500 then a message will be displayed as shown in the above output.

Conclusion:
There is a default response for each signal. The action that a script or program executes in response to a signal is known as the default action for a signal. For example- stop the procedure, disregard the signal, stop the operation, carry on a halted procedure, etc.