Linux Signals Fundamentals – Part I

*by* HIMANSHU ARORA *on* MARCH 5, 2012

What is a signal? Signals are *software interrupts*.

A robust program need to handle signals. This is because signals are a way to deliver asynchronous events to the application.

A user hitting ctrl+c, a process sending a signal to kill another process etc are all such cases where a process needs to do signal handling.

### Linux Signals

In Linux, every signal has a name that begins with characters SIG. For example:

* A SIGINT signal that is generated when a user presses ctrl+c. This is the way to terminate programs from terminal.
* A SIGALRM is generated when the timer set by alarm function goes off.
* A SIGABRT signal is generated when a process calls the abort function.
* etc

When the signal occurs, the process has to tell the kernel what to do with it.  There can be three options through which a signal can be disposed:

1. The signal can be ignored. By ignoring we mean that nothing will be done when signal occurs. Most of the signals can be ignored but signals generated by hardware exceptions like divide by zero, if ignored can have weird consequences. Also, a couple of signals like SIGKILL and SIGSTOP cannot be ignored.
2. The signal can be caught. When this option is chosen, then the process registers a function with kernel. This function is called by kernel when that signal occurs. If the signal is nonfatal for the process then in that function the process can handle the signal properly or otherwise it can chose to terminate gracefully.
3. Let the default action apply. Every signal has a default action. This could be process terminate, ignore etc.

As we already stated that two signals SIGKILL and SIGSTOP cannot be ignored. This is because these two signals provide a way for root user or the kernel to kill or stop any process in any situation .The default action of these signals is to terminate the process. Neither these signals can be caught nor can be ignored.

**What Happens at Program Start-up?**

It all depends on the process that calls exec. When the process is started the status of all the signals is either ignore or default. It’s the later option that is more likely to happen unless the process that calls exec is ignoring the signals.

It is the property of exec functions to change the action on any signal to be the default action. In simpler terms, if parent has a signal catching function that gets called on signal occurrence then  if that parent execs a new child process, then this function has no meaning in the new process and hence the disposition of the same signal is set to the default in the new process.

Also, Since we usually have processes running in background so the shell just sets the quit signal disposition as ignored since we do not want the background processes to get terminated by a user pressing a ctrl+c key because that defeats the purpose of making a process run in background.

**Why Signal Catching Functions should be Reentrant?**

As we have already discussed that one of the option for signal disposition is to catch the signal. In the process code this is done by registering a function to kernel which the kernel calls when the signal occurs. One thing to be kept in mind is that the function that the process registers should be *reentrant*.

Before explaining the reason, let’s first understand what are reentrant functions? A *reentrant function* is a function whose execution can be stopped in between due to any reason (like due to interrupt or signal) and then can be reentered again safely before its previous invocations complete the execution.

Now coming back to the issue, suppose a function func() is registered for a call back on a signal occurrence. Now assume that this func() was already in  execution when the signal occurred. Since this function is call back for this signal so the current execution on this signal will be stopped by the scheduler and this function will be called again (due to signal).

The problem can be if func() works on some global values or data structures that are left in inconsistent state when the execution of this function was stopped in middle then the second call to same function(due to signal) may cause some undesired results.

So we say that signal catching functions should be made reentrant.

Refer to our articles [send-signal-to-process](https://www.thegeekstuff.com/2011/02/send-signal-to-process/) and [Linux fuser command](https://www.thegeekstuff.com/2012/02/linux-fuser-command/) to see practical examples on how to send signals to a process.

**Threads and Signals**

We already saw in the one of previous sections that signal handling comes with its own bit of complexity (like using reentrant functions) . To add on to the complexity, we usually have multi threaded applications where signal handling becomes really complicated.

Every thread has its own private signal mask(a mask that defines which signals are deliverable) but the way signal disposition is done is shared by all the threads in the application. This means that a disposition for a particular signal set by a thread can easily be overruled by some other thread. In this case the disposition mechanism changes for all the threads.

For example, a thread A can choose to ignore a particular signal but a thread B in the same process can choose to catch the same signal by registering a callback function to the kernel. In this case the request made by thread A gets overruled by thread B’s request.

Signals are delivered only to a single thread in any process. Apart from the the hardware exceptions or the timer expiry (which are delivered to thread which caused the event) all the signals are passed to the process arbitrarily.

To counter this shortcoming there are some posix APIs like pthread\_sigmask() etc that can be used.

In the [next article (part 2)](https://www.thegeekstuff.com/2012/03/catch-signals-sample-c-code/) of this series, we will discuss about how to catch signals in a process, and explain the practical aspect of signal handling using code snippets.

# Linux Signals – Example C Program to Catch Signals (SIGINT, SIGKILL, SIGSTOP, etc.)

*by* HIMANSHU ARORA *on* MARCH 9, 2012

In the part 1 of the Linux Signals series, we learned about the [fundamental concepts behind Linux signals](https://www.thegeekstuff.com/2012/03/linux-signals-fundamentals/).

Building on the previous part, in this article we will learn about how to catch signals in a process. We will present the practical aspect of signal handling using C program code snippets.

### 1) Catching a Signal

As already discussed in the previous article, if a process wishes to handle certain signals then in the code, the process has to register a signal handling function to the kernel.

The following is the prototype of a signal handling function:

void <signal handler func name> (int sig)

The signal handler function has void return type and accepts a signal number corresponding to the signal that needs to be handled.

To get the signal handler function registered to the kernel, the signal handler function pointer is passed as second argument to the ‘signal’ function. The prototype of the signal function is :

void (\*signal(int signo, void (\*func )(int)))(int);

This might seems a complicated declaration. If we try to decode it :

* The function requires two arguments.
* The first argument is an integer (signo) depicting the signal number or signal value.
* The second argument is a pointer to the signal handler function that accepts an integer as argument and returns nothing (void).
* While the ‘signal’ function itself returns function pointer whose return type is void.

Well, to make things easier, lets use typedef :

typedef void sigfunc(int)

So, here we have made a new type ‘sigfunc’.  Now using this typedef, if we redesign the prototype of the signal handler :

sigfunc \*signal(int, sigfunc\*);

Now we see that its easier to comprehend that the signal handler function accepts an integer and a sigfunc type function pointer while it returns a sigfunc type function pointer.

### Example C Program to Catch a Signal

Most of the Linux users use the key combination Ctr+C to terminate processes in Linux.

Have you ever thought of what goes behind this. Well, whenever ctrl+c is pressed, a signal SIGINT is sent to the process. The default action of this signal is to terminate the process. But this signal can also be handled. The following code demonstrates this :

#include<stdio.h>

#include<signal.h>

#include<unistd.h>

void sig\_handler(int signo)

{

if (signo == SIGINT)

printf("received SIGINT\n");

}

int main(void)

{

if (signal(SIGINT, sig\_handler) == SIG\_ERR)

printf("\ncan't catch SIGINT\n");

// A long long wait so that we can easily issue a signal to this process

while(1)

sleep(1);

return 0;

}

In the code above, we have simulated a long running process using an infinite while loop.

A function sig\_handler is used a s a signal handler. This function is registered to the kernel by passing it as the second argument of the system call ‘signal’ in the main() function. The first argument to the function ‘signal’ is the signal we intend the signal handler to handle which is SIGINT in this case.

On a side note, the use of function sleep(1) has a reason behind. This function has been used in the while loop so that while loop executes after some time (ie one second in this case). This becomes important because otherwise an infinite while loop running wildly may consume most of the CPU making the computer very very slow.

Anyways, coming back , when the process is run and we try to terminate the process using Ctrl+C:

$ ./sigfunc

^Creceived SIGINT

^Creceived SIGINT

^Creceived SIGINT

^Creceived SIGINT

^Creceived SIGINT

^Creceived SIGINT

^Creceived SIGINT

We see in the above output that we tried the key combination ctrl+c several times but each time the process didn’t terminate. This is because the signal was handled in the code and this was confirmed from the print we got on each line.

### SIGKILL, SIGSTOP and User Defined Signals

Apart from handling the standard signals(like INT, TERM etc) that are available. We can also have user defined signals that can be sent and handled. Following is the code handling a user defined signal USR1 :

#include<stdio.h>

#include<signal.h>

#include<unistd.h>

void sig\_handler(int signo)

{

if (signo == SIGUSR1)

printf("received SIGUSR1\n");

else if (signo == SIGKILL)

printf("received SIGKILL\n");

else if (signo == SIGSTOP)

printf("received SIGSTOP\n");

}

int main(void)

{

if (signal(SIGUSR1, sig\_handler) == SIG\_ERR)

printf("\ncan't catch SIGUSR1\n");

if (signal(SIGKILL, sig\_handler) == SIG\_ERR)

printf("\ncan't catch SIGKILL\n");

if (signal(SIGSTOP, sig\_handler) == SIG\_ERR)

printf("\ncan't catch SIGSTOP\n");

// A long long wait so that we can easily issue a signal to this process

while(1)

sleep(1);

return 0;

}

We see that in the above code, we have tried to handle a user defined signal USR1. Also, as we know that two signals KILL and STOP cannot be handled. So we have also tried to handle these two signals so as to see how the ‘signal’ system call responds in this case.

When we run the above code :

$ ./sigfunc

can't catch SIGKILL

can't catch SIGSTOP

So the above output makes clear that as soon as the system call ‘signal’ tries to register handler for KILL and STOP signals, the signal function fails indicating that these two signals cannot be caught.

Now we try to pass the signal USR1 to this process using the [kill command](https://www.thegeekstuff.com/2009/12/4-ways-to-kill-a-process-kill-killall-pkill-xkill/):

$ kill -USR1 2678

and on the terminal where the above program is running we see :

$ ./sigfunc

can't catch SIGKILL

can't catch SIGSTOP

received SIGUSR1

So we see that the user defined signal USR1 was received in the process and was handled properly.