SIGNALS AND ALARM

**Concepts and Overview**

A signal is a notification to a process that an event has occurred. Signals are software interrupts. Most nontrivial application programs need to deal with signals. Signals are analogous to hardware interrupts in that they interrupt the normal flow of execution of a program; in most cases, it is not possible to predict exactly when a signal will arrive. Signals provide a way of handling asynchronous events—for example, a user at a terminal typing the interrupt key to stop a program or the next program in a pipeline terminating prematurely. when a signal is delivered to a process, the process will stop what its doing, either handle or ignore the signal, or in some cases terminate, depending on the signal.

Signals also are delivered in an unpredictable way out of sequence with the program because signals usually originate outside of the current executing process. Another way to view signals is a mechanism for handling **asynchronous events**. As opposed to **synchronous events**, which is when a standard program executing iteratively, one line of code following another, **asynchronous events** is when portions of the program may execute out of order, or not immediately in a iterative style. Asynchronous events are typically due to external events at the interaction layer between the hardware and the operating system; the signal, itself, is the way for the operating system to communicate these events to the processes

One process can (if it has suitable permissions) send a signal to another process. In this use, signals can be employed as a synchronization technique, or even as a primitive form of interprocess communication (IPC). It is also possible for a process to send a signal to itself. However, the usual source of many signals sent to a process is the kernel. Among the types of events that cause the kernel to generate a signal for a process are the following:

* A hardware exception occurred, meaning that the hardware detected a fault condition that was notified to the kernel, which in turn sent a corresponding signal to the process concerned. Examples of hardware exceptions include executing a malformed machine-language instruction, dividing by 0, or referencing a part of memory that is inaccessible. See sample programs at the end of these document.
* The user typed one of the terminal special characters that generate signals. These characters include the interrupt character (usually Control-C) and the suspend character (usually Control-Z).
* A software event occurred. For example, input became available on a file descriptor, the terminal window was resized, a timer went off, the process’s CPU time limit was exceeded, or a child of this process terminated.

First, every signal has an unique name and all begin with the three characters SIG. For example, SIGABRT is the abort signal that is generated when a process calls the abort function.  SIGALRM is the alarm signal that is generated when the timer set by the alarm function goes off.  When the user types the interrupt character, SIGINT (signal  number 2) is delivered to a process. Each signal name is defined as a unique (small) integer, starting sequentially from 1. These integers are defined in <signal.h> with symbolic names of the form SIGxxxx.  Since the actual numbers used for each signal vary across implementations, it is these symbolic names that are always used in programs.

Signals fall into two broad categories. The first set constitutes the traditional or standard signals, which are used by the kernel to notify processes of events. On Linux, the standard signals are numbered from 1 to 31. We describe the standard signals in this document.

DO NOT DISTURB : Can a running process block signals ?

Between the time a signal is generated externally and the time it is delivered to the process, a signal is said to be pending. Normally, a pending signal is delivered to a process as soon as it is next scheduled to run, or immediately if the process is already running (e.g., if the process sent a signal to itself). Sometimes, however, we need to ensure that a segment of code is not interrupted by the delivery of a signal. To do this, we can add a signal to the process’s signal mask—a set of signals whose delivery is currently blocked. If a signal is generated while it is blocked, it remains pending until it is later unblocked (removed from the signal mask). Various system calls allow a process to add and remove signals from its signal mask. This is much like "DO NOT DISTURB" sign you see in hotel rooms.

SIGNAL DELIVERED:

Upon delivery of a signal, a process carries out one of the following default actions, **depending on the signal**:

* The signal is ignored; that is, it is discarded by the kernel and has no effect on the process. (The process never even knows that it occurred.)
* The process is terminated (killed). This is sometimes referred to as abnormal process termination, as opposed to the normal process termination that occurs when a process terminates using exit().
* A core dump file is generated, and the process is terminated. A core dump file contains an image of the virtual memory of the process, which can be loaded into a debugger in order to inspect the state of the process at the time that it terminated.
* The process is stopped—execution of the process is suspended.
* Execution of the process is resumed after previously being stopped.

Instead of accepting the default for a particular signal, a program can change the action that occurs when the signal is delivered. This is known as setting the disposition of the signal. A program can set one of the following dispositions for a signal:

* The default action should occur. This is useful to undo an earlier change of the disposition of the signal to something other than its default.
* The signal is ignored. This is useful for a signal whose default action would be to terminate the process.
* A signal handler is executed.

What is a signal handler ?  A signal handler is a function, written by the programmer, that performs appropriate tasks in response to the delivery of a signal. For example, the shell has a handler for the SIGINT signal (generated by the interrupt character, Control-C) that causes it to stop what it is currently doing and return control to the main input loop, so that the user is once more presented with the shell prompt. Notifying the kernel that a handler function should be invoked is usually referred to as installing or establishing a signal handler. When a signal handler is invoked in response to the delivery of a signal, we say that the signal has been handled or, synonymously, caught.

**Signal Types and Default Actions**

Earlier, we mentioned that the standard signals are numbered from 1 to 31 on  Linux. However, the Linux signal(7) manual page lists more than 31 signal names. The excess names can be accounted for in a variety of ways. Some of the names are simply synonyms for other names, and are defined for source compatibility with other UNIX implementations. Other names are defined but unused. The following list describes the various signals:

|  |  |
| --- | --- |
| SIGABRT | A process is sent this signal when it calls the abort() function.  By default, this signal terminates the process with a core dump. This achieves the intended purpose of the abort() call: to produce a core dump for debugging. |
| SIGALRM | The kernel generates this signal upon the expiration of a real-time timer set by a call to alarm()or setitimer(). A real-time timer is one that counts  according to wall clock time (i.e., the human notion of elapsed time). |
| SIGBUS | This signal (“bus error”) is generated to indicate certain kinds of memory access  errors. One such error can occur when using memory mappings created with mmap(), if we attempt to access an address that lies beyond the end of the underlying memory-mapped file |
| SIGCHLD | This signal is sent (by the kernel) to a parent process when one of its children terminates (either by calling exit() or as a result of being killed by a signal). It may also be sent to a process when one of its children is stopped or resumed by a signal. |
| SIGCONT | When sent to a stopped process, this signal causes the process to resume (i.e., to be rescheduled to run at some later time). When received by a process that is not currently stopped, this signal is ignored by default. A process may catch this signal, so that it carries out some action when it resumes |
| SIGFPE | This signal is generated for certain types of arithmetic errors, such as divide-by-zero. The suffixFPE is an abbreviation for floating-point exception, although this signal can also be generated for integer arithmetic errors. The precise details of when this signal is generated depend on the hardware architecture and the settings of CPU control registers. For example, on x86-32, integer divide-by-zero always yields a SIGFPE, |
| SIGINT | When the user types the terminal interrupt character (usually Control-C), the terminal driver sends this signal to the foreground process group. The default action for this signal is to terminate the process. |
| SIGKILL | This is the sure kill signal. It can’t be blocked, ignored, or caught by a handler, and thus always terminates a process. |
| SIGPIPE | This signal is generated when a process tries to write to a pipe, a FIFO, or a socket for which there is no corresponding reader process. This normally occurs because the reading process has closed its file descriptor for the IPC channel |
| SIGQUIT | When the user types the quit character (usually Control-\) on the keyboard, this signal is sent to the foreground process group. By default, this signal terminates a process and causes it to produce a core dump, which can then be used for debugging. Using SIGQUIT in this manner is useful with a program that is stuck in an infinite loop or is otherwise not responding. By typingControl-\ and then loading the resulting core dump with the gdb debugger and using thebacktrace command to obtain a stack trace, we can find out which part of the program code was executing. |
| SIGTERM | This is the standard signal used for terminating a process and is the default  signal sent by thekill and killall commands. Users sometimes explicitly send the SIGKILL signal to a process usingkill –KILL or kill –9. However, this is generally a mistake. A well-designed application will have a handler for SIGTERM that causes the application to exit gracefully, cleaning up temporary files and releasing other resources beforehand. Killing a process with SIGKILL bypasses theSIGTERM handler. Thus, we should always first attempt to terminate a process using SIGTERM, and reserve SIGKILL as a last resort for killing runaway processes that don’t respond toSIGTERM |
| SIGTSTP | This is the job-control stop signal, sent to stop the foreground process group when the user types the suspend character (usually Control-Z) on the keyboard. |
|  |  |

 The #define values are coded in <sys/signal.h> and some of the values are

#define SIGHUP 1 /\* hangup \*/

#define SIGINT 2 /\* interrupt \*/

#define SIGQUIT 3 /\* quit \*/

#define SIGILL 4 /\* illegal instruction \*/

#define SIGABRT 6 /\* abort() \*/

#define SIGFPE 8 /\* floating point exception \*/

#define SIGKILL 9 /\* kill (cannot be caught or ignored) \*/

#define SIGBUS 10 /\* bus error \*/

#define SIGSEGV 11 /\* segmentation violation \*/

#define SIGSYS 12 /\* bad argument to system call \*/

#define SIGPIPE 13 /\* write on a pipe with no one to read it \*/

#define SIGALRM 14 /\* alarm clock \*/

#define SIGTERM 15 /\* software termination signal from kill \*/

#define SIGURG 16 /\* urgent condition on IO channel \*/

#define SIGSTOP 17 /\* sendable stop signal not from tty \*/

#define SIGCHLD 20 /\* to parent on child stop or exit \*/

#define SIGWINCH 28 /\* window size changes \*/

#define SIGUSR1 30 /\* user defined signal 1 \*/

#define SIGUSR2 31 /\* user defined signal 2 \*/

**Implementation of Signal Handlers**

A signal handler (also called a signal catcher) is a function that is called when a specified

signal is delivered to a process. Invocation of a signal handler may interrupt the main program flow at any time; the kernel calls the handler on the process’s behalf, and when the handler returns, execution of the program resumes at the point where the handler interrupted it. Although signal handlers can do virtually anything, they should, in general, be designed to be as simple as possible.

SIGINT

#include <signal.h>

#include <unistd.h>

#include <stdio.h>

In the main function, we have registered our signal handler controlC with our operating system using the signal function. This main program will be in an infinite loop. In the signal handler, we are printing a statement and returning immediately. Generally, we don't print in the signal handlers, but for our purposes, we will use a printf statement. The parameter signum is not utilized though the OS will pass us the value.

Some key points to note from this program is that the second argument to signal() is a function pointer, a reference to a function to call. This tells the operating system that whenever this signal is sent to this process, run this function as the signal handler.

Also, the execution of the signal handler is asynchronous, which means the current state of the program will be paused while the signalhandler executes, and then execution will resume from the pause point, much like context switching.

void controlC ( int signum)

{

printf ( "Thank you for pressing control-C\n");

}

int main ( )

{

signal ( SIGINT, controlC) ; // registering our signal handler function with OS

while ( 1 ) ; // infinite loop

exit ( 0 ) ;

}

Compile the above program and run. This program will be in an infinite loop. To see this signal in action, press control-c . The shell passes the control-C key stroke to the Operating system, OS will call the signal handler. Remember, before the function will be executed, our program is executing the while loop. The while loop will be stopped, the signal handler will be executing. When the handler finishes the printf statement, the control goes back to the while loop.

Now, if you prefer to exit the program in the handler, you can insert exit statement.

void controlC ( int signum)

{

printf ( "Thank you for pressing control-C\n");

exit ( 0 ) ;

}

SIGQUIT

void controlQuit ( int signum)

{

printf ( "Thank you for pressing control-backslash \n");

exit ( 0 ) ;

}

int main ( )

{

signal( SIGQUIT, controlQuit);

while ( 1 ) ;

exit ( 0 ) ;

}

In the above program, we are registering a signal handler for SIGQUIT signal. The main program will be infinite loop. To generate the signal, press control \ ( backslash ) . You will see the program printing the statement and quitting the program.

SIGWINCH

As the name implies, this signal is sent whenever the size of the WINdow is CHanged.

void windowChange ( int signum)

{

printf ( "Window size changed\n");

}

int main ( )

{

signal( SIGWINCH, windowChange ) ;

while ( 1 ) ;

exit ( 0 ) ;

}

In the above program, we are registering a signal handler to receive signals whenever the size of the Window Changes. When you run the program, try to change the size of the window by resizing it. Note: We are exiting out of the handler in this program.

SIGCHD

void childExit ( int signum )

{

printf ( "Child process Terminated \n");

exit ( 0 ) ;

}

int main ( )

{

signal ( SIGCHLD, childExit ) ;

int child = fork () ;

if ( child == 0 ) // child

{

sleep (2); // when the child dies after 2 seconds

// parent will be notified.

}

else {

while ( 1 ) ;

}

exit ( 0 ) ;

}

In the above program, we are registering a signal for the parent process to receive a signal when the child process terminates. In the main program, we are forking a child process. The child process sleep for 2 seconds and dies. When it dies, the parent process gets a signal. Just compile and run. In 2 seconds, you should see the statement in the handler function.

SIGSEGV

void sigsegv ( int signum )

{

printf ( "illegal memory access \n");

exit ( 1 );

}

int main ( )

{

signal ( SIGSEGV, sigsegv ) ;

int x = 10;

scanf ( "%d", x); // it should &x

exit ( 0 ) ;

}

In the above program, we are registering a signal handler to handle any illegal memory access. In the scanf function, we intentionally removed the address & for x and we are passing the address = 10. When you run program, the system call will place the value entered by the user in address 10 creating an illegal memory access. The kernel will generate a signal and our handler will print and exit.

SIGFPE

void sigfpe ( int signum)

{

printf ( "Floating Point Exception \n");

exit ( 1 );

}

int main ( )

{

signal( SIGFPE, sigfpe);

int x = 1/0;

exit ( 0 ) ;

}

SIGFPE is a signal for floating point exception. When you divide a number by zero, floating point exception signal is generated.

SIGPIPE

void sigpipe ( int signum )

{

printf ( "child may have died, and Pipe broken \n");

exit ( 0 ) ;

}

int main(void)

{

int fd[2] ;

pipe(fd);

signal ( SIGPIPE, sigpipe);

switch ( fork() )

{

case 0 : /\* child \*/

close(fd[1]);

close(fd[0]); /\* closing read end of pipe \*/

exit(0);

default: /\* parent \*/

sleep ( 1);

close(fd[0]);

write(fd[1], "ABCD\n", 5);

}

return(0);

}

In the above program, we create a pipe and we close the pipes on the child process. When the parent writes on the pipe, because the child is already terminated, a SIGPIPE signal is generated. You need sleep on the parent process to give enough time for child to terminate .

**SIGSTOP** The SIGSTOP signal stops the process. It cannot be handled, ignored, or blocked.

The SIGKILL signal is used to cause immediate program termination. It cannot be handled or ignored, and is therefore always fatal. It is also not possible to block this signal.

**GENERATING SIGNALS FROM COMMAND LINE**

So far we have seen signals were triggered in our program. Now we will try to trigger from command line of another shell .

In one shell window, try to run this program from the command line.

void sigsegv ( int signum )

{

printf ( "illegal memory access \n");

exit ( 1 );

}

int main ( )

{

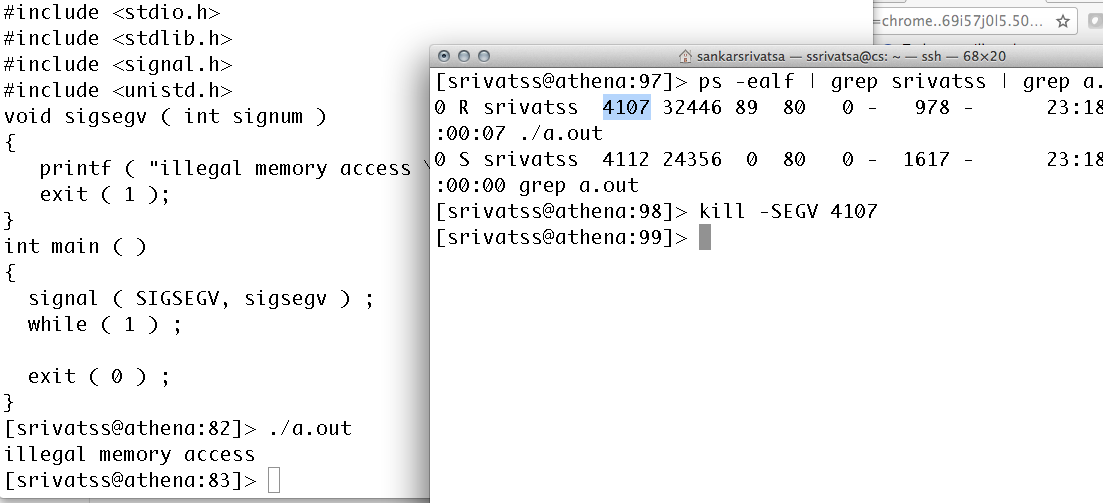
signal ( SIGSEGV, sigsegv ) ;

while ( 1 ) ;

exit ( 0 ) ;

}

Then, you need to know the process ID of the above program running.



In the above diagram, the process ID for the program 4835 .

In another shell terminal, send the signal to the process running 4835 using the command

kill -SEGV 4835

or

kill -11 4835

The 11 above is the constant value defined for SIGSEGV in the <sys/signal.h>

You can also use killall command , instead of kill, but use the name of the program.

killall -SIGKILL a.out

Alarms :

A SIGALRM signal is an alarm set by the function alarm ( int sec ) after sec seconds have elapsed since calling the function, it is delivered by the Operating System much like signals we read so far.

void buzzMe( int signum )

{

printf ( "Wake up...\n");

}

int main()

{

signal ( SIGALRM, buzzMe);

alarm ( 2 );

while ( 1 ) ;

}

The above looks very similar to the program we wrote earlier. But here, we use SIGALRM signal value to register our alarm with the operating system and the alarm handler is defined. Then, we have to request an alarm , here we use 2 seconds. After the request has been put, we wait using a while loop. You can instead use a function pause or sleep , but these are implementation specific , in general avoid using signals and sleep.

Because only one alarm can used in a program, we cannot set another alarm. In case you plan to use another alarm, the previous alarm will be replaced.