LINUX JOURNAL

Linux Signals for the Application Programmer

Date: Saturday, March 01, 2003
Topic: Embedded

Dr B. Thangaraju

Signals are a fundamental method for interprocess communication ad are used in everything from network servers to media players. Here's how you can use them in your applications.

A good understanding of signals is important for an application programmer working in the Linux environment. Knowledge of the signaling mechanism and familiarity with signal-related functions help one write programs more efficiently.

An application program executes sequentially if every instruction runs properly. In case of an error or any anomaly during the execution of a program, the kernel can use signals to notify the process. Signals also have been used to communicate and synchronize processes and to simplify interprocess communications (IPCs). Although we now have advanced synchronization tools and many IPC mechanisms, signals play a vital role in Linux for handling exceptions and interrupts. Signals have been used for approximately 30 years without any major modifications.

The first 31 signals are standard signals, some of which date back to 1970s UNIX from Bell Labs. The POSIX (Portable Operating Systems and Interface for UNIX) standard introduced a new class of signals designated as real-time signals, with numbers ranging from 32 to 63.

A signal is generated when an event occurs, and then the kernel passes the event to a receiving process. Sometimes a process can send a signal to other processes. Besides process-to-process signaling, there are many situations when the kernel originates a signal, such as when file size exceeds limits, when an I/O device is ready, when encountering an illegal instruction or when the user sends a terminal interrupt like Ctrl-C or Ctrl-Z.

Every signal has a name starting with SIG and is defined as a positive unique integer number. In a shell prompt, the `kill -l` command will display all signals with signal number and corresponding signal name. Signal numbers are defined in the /usr/include/bits/signum.h file, and the source file is /usr/src/linux/kernel/signal.c.

A process will receive a signal when it is running in user mode. If the receiving process is running in kernel mode, the execution of the signal will start only after the process returns to user mode.

Signals sent to a non-running process must be saved by the kernel until the process resumes execution. Sleeping processes can be interruptible or uninterruptible. If a process receives a signal when it is in an interruptible sleep state, for example, waiting for terminal I/O, the kernel will awaken the process to handle the signal. If a process receives a signal when it is in uninterruptible sleep, such as waiting for disk I/O, the kernel defers the signal until the event completes.

When a process receives a signal, one of three things could happen. First, the process could ignore the signal. Second, it could catch the signal and execute a special function called a signal handler. Third, it could execute the default action for that signal; for example, the default action for signal 15, SIGTERM, is to terminate the process. Some signals cannot be ignored, and others do not have default actions, so they are ignored by default. See the signal(7) man page for a reference list of signal names, numbers, default actions and whether they can be caught.

When a process executes a signal handler, if some other signal arrives the new signal is blocked until the handler returns. This article explains the fundamentals of the signaling mechanism and elaborates on signal-related functions with syntax and working procedures.

Signals inside the Kernel

Where is the information about a signal stored in the process? The kernel has a fixed-size array of proc structures called the process table. The u or user area of the proc structure maintains control information about a process. The major fields in the u area include signal handlers and related information. The signal handler is an array with each element for each type of signal being defined in the system, indicating the action of the process on the receipt of the signal. The proc structure maintains signal-handling information, such as masks of signals that are ignored, blocked, posted and handled.

Once a signal is generated, the kernel sets a bit in the signal field of the process table entry. If the signal is being ignored, the kernel returns without taking any action. Because the signal field is one bit per signal, multiple occurrences of the same signal are not maintained.

When the signal is delivered, the receiving process should act depending on the signal. The action may be terminating the process, terminating the process after creating a core dump, ignoring the signal, executing the user-defined signal handler (if the signal is caught by the process) or resuming the process if it is temporarily suspended.

The core dump is a file called core, which has an image of the terminated process. It contains the process' variables and stack details at the time of failure. From a core file, the programmer can investigate the reason for termination using a debugger. The word core appears here for a historical reason: main memory used to be made from doughnut-shaped magnets called inductor cores.

Catching a signal means instructing the kernel that if a given signal has occurred, the program's own signal handler should be executed, instead of the default. Two exceptions are SIGKILL and SIGSTOP, which cannot be caught or ignored.

sigset_t is a basic data structure used to store the signals. The structure sent to a process is a sigset_t array of bits, one for each signal type:

```
typedef struct {
                unsigned long sig[2];
             }  sigset_t;
```

Because each unsigned long number consists of 32 bits, the maximum number of signals that may be declared in Linux is 64 (according to POSIX compliance). No signal has the number 0, so the other 31 bits in the first element of sigset_t are the standard first 31 signals, and the bits in the second element are the real-time signal numbers 32-64. The size of sigset_t is 128 bytes.

Handling Signals

There are many system calls and signal-supported library functions, which provide an easy and efficient way of handling the signals in a process. We start with the standard old signal system call, then we discuss some useful functions like sigaction, sigaddset, sigemptyset, sigdelset, sigismember and kill.

The Signal System Call

The signal system call is used to catch, ignore or set the default action of a specified signal. It takes two arguments: a signal number and a pointer to a user-defined signal handler. Two reserved predefined signal handlers are available in Linux: SIG_IGN and SIG_DFL. SIG_IGN will ignore a specified signal, and SIG_DFL will set the signal handler to the default action for that signal (see `man 2 signal`).

On success, the system call returns the previous value of the signal handler for the specified signal. If the signal call fails, it returns SIG_ERR. Listing 1 explains how to catch, ignore and set the default action of SIGINT. Try pressing Ctrl-C, which sends SIGINT, during each part.

Listing 1. Catching and Ignoring a Signal

```
#include <signal.h>

void my_handler (int sig); /* function prototype */

int main ( void ) {
```

```
/* Part I: Catch SIGINT */
    signal (SIGINT, my_handler);
    printf ("Catching SIGINT\n");
    sleep(3);
    printf (" No SIGINT within 3 seconds\n");

/* Part II: Ignore SIGINT */
    signal (SIGINT, SIG_IGN);
    printf ("Ignoring SIGINT\n");
    sleep(3);
    printf ("No SIGINT within 3 seconds\n");

/* Part III: Default action for  SIGINT */
    signal (SIGINT, SIG_DFL);
    printf ("Default action for SIGINT\n");
    sleep(3);
    printf ("No SIGINT within 3 seconds\n");
    return 0;
}

/* User-defined signal handler function */
void my_handler (int sig) {
    printf ("I got SIGINT, number %d\n", sig);
    exit(0);
}
```

sigaction

The sigaction system call can be used instead of signal because it has lot of control over a given signal. The syntax of sigaction is:

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

The first argument, signum, is a specified signal; the second argument, sigaction, is used to set the new action of the signal signum; and the third argument is used to store the previous action, usually NULL.

The sigaction structure is defined as:

```
struct sigaction {
    void (*sa_handler)(int);
    void (*sa_sigaction)(int, siginfo_t *, void *);
    sigset_t sa_mask;
    int sa_flags;
}
```

The members of the sigaction structure are described as follows.

sa_hander: a pointer to a user-defined signal handler or predefined signal handler (SIG_IGN or SIG_DFL).

sa_mask: specifies a mask of signals when the signal is handled. To avoid the blocking of signals, the SA_NODEFER or SA_NOMASK flags can be used.

sa_flags: specifies the action of signal. Sets of flags are available for controlling the signal in a different manner. More than one flag can be used by ORing:

- SA_NOCLDSTOP: if we specify the SIGCHLD signal, when the child has stopped its execution it does not receive notification.
- SA_ONESHOT or SA_RESETHAND: restores the default action of the signal after the user-defined signal handler is executed. To avoid setting the default action, SA_RESTART can be used.
- SA_NOMASK or SA_NODEFER prevents masking the signal. SA_SIGINFO is used to receive signal-related information.

sa_sigaction: if the SA_SIGINFO flag is used in sa_flags, instead of specifying the signal handler in sa_handler, sa_sigaction should be used.

sa_sigaction is a pointer to a function that takes three arguments, not one as sa_handler does, for example:

```
void my_handler (int signo, siginfo_t *info,
                     void *context)
```

Here, signo is the signal number, and info is a pointer to the structure of type siginfo_t, which specifies the signal-related information; and context is a pointer to an object of type ucontext_t, which refers to the receiving process context that was interrupted with the delivered signal.

Listing 2 is similar to Listing 1 but uses the sigaction system call instead of the signal system call. Listing 3 explains signal-related information using the SIG_INFO flag.

Listing 2. Same as Listing 1, but with sigaction

```
#include <signal.h>
#include <stdio.h>

void my_handler (int sig); /* function prototype */

int main ( void ) {

    struct sigaction my_action;
/* Part I: Catch SIGINT */

    my_action.sa_handler = my_handler;
    my_action.sa_flags = SA_RESTART;
    sigaction (SIGINT, &my_action, NULL);
    printf ("Catching SIGINT\n");
    sleep(3);
    printf (" No SIGINT within 3 seconds\n");


/* Part II: Ignore SIGINT */

   my_action.sa_handler = SIG_IGN;
   my_action.sa_flags = SA_RESTART;
   sigaction (SIGINT, &my_action, NULL);
   printf ("Ignoring SIGINT\n");
   sleep(3);
   printf (" Sleep is over\n");


/* Part III: Default action for  SIGINT */

  my_action.sa_handler = SIG_DFL;
  my_action.sa_flags = SA_RESTART;
  sigaction (SIGINT, &my_action, NULL);
  sleep(3);
  printf ("No SIGINT within 3 seconds\n");
}

void my_handler (int sig) {
    printf ("I got SIGINT, number %d\n", sig);
    exit(0);
}
```

Listing 3. Using SA_SIGINFO and sa_sigaction to

Extract Information from a Signal

```
#include <unistd.h>
#include <sys/types.h>
#include <signal.h>
#include <bits/siginfo.h>
#include <stdio.h>

void handler (int signo, siginfo_t *info,
              void *context);

main () {

   struct sigaction my_action;

   my_action.sa_flags = SA_SIGINFO;
   my_action.sa_sigaction = handler;
   sigaction(SIGINT, &my_action, NULL);

   printf ("Catching SIGINT\n");
   sleep(5);
   printf ("Done.\n");
}

void handler (int signo, siginfo_t *info,
              void *context)
 {
    printf ("Signal number: %d\n", info->si_signo);

 /* Elements of the siginfo_t structure are listed
    in man 2 sigaction.
 */
}
```

Sending Signals

Until now, we've been pressing Ctrl-C to send SIGINT from the shell. To do it from a program, use the kill system call, which accepts two arguments, process ID and signal number:

```
int kill ( pid_t process_id, int signal_number );
```

If the pid is positive, the signal is sent to a particular process. If the pid is negative, the signal is sent to the process whose group ID matches the absolute value of pid.

As you might expect, the kill command, which exists as a standalone program (/bin/kill) and is also built into bash (try `help kill`) uses the kill system call to send a signal.

Not all processes can send signals to each other. In order for one process to send a signal to another, either the sender must be running as root, or the sender's real or effective user ID must be the same as the real or saved ID of the receiver. This means your shell, running as you, can signal a setuid program that you started, but that is now running as root, for example:

```
cp /bin/sleep ~/rootsleep
sudo chmod u+s ~/rootsleep
./rootsleep 40
killall rootsleep
rm ~/rootsleep
```

A normal user can't send signals to system processes such as swapper and init.

You also can use kill to find out if a process exists. Specify a signal number of 0, and if the process exists, the kill returns zero; if it doesn't, kill returns -1.

Listing 4. Programs to Send and Receive SIGINT

```
#include <signal.h>

main ( ) {
    int process_id;
    printf ("Enter process_id which you want "
            "to send a signal : ");
    scanf ("%d", &process_id);

   if (!(kill ( process_id, SIGINT)))
       printf ("SIGINT sent to %d\n", process_id);
   else if (errno == EPERM)
       printf ("Operation not permitted.\n");
   else
       printf ("%d doesn't exist\n", process_id);
}

/* Listing 4a. This program will run until it
   receives SIGINT */

#include <signal.h>

 main ( ) {
   printf (" This process id is %d. "
   "Waiting for SIGINT.\n", getpid());
   for (;;);
}
```

Listings 4 and 4a explain how to use the kill system call. First, execute the 4a program in one window and get its process ID. Now, run the Listing 4 program in another window and give the 4a example's pid as the input.

This article should help you understand the fundamental concept of a signal and some of its importance. Try the sample programs, and see the man pages for the system calls and the references in Resources for more information.

Resources

Advanced Programming in the UNIX Environment by W. Richard Stevens, Pearson Education Asia, 1993, pp. 263-324.

Linux Application Development by Michael K. Johnson and Erik W. Troan, Addison-Wesley, 1998.

``The Linux Signals Handling Model" by Moshe Bar, *Linux Journal*, May 01, 2000 (available at www.linuxjournal.com/article/3985).

Understanding the Linux Kernel by D. P. Bovet and M. Cesati, O'Reilly & Associates, 1998, pp. 233-248.

***Dr B. Thangaraju** received a PhD in Physics and worked as a research associate for five years at the Indian Institute of Science, India. He is presently working as a manager at Talent Transformation, Wipro Technologies, India. He has published many research papers in renowned international journals. His current areas of research, study and knowledge dissemination are the Linux kernel, device drivers and real-time Linux.*