Signals and Interrupt Handlers

Mark Sitkowski
Design Simulation Systems Ltd
www.designsim.com.au

So what is a signal?

A Unix signal, is a software interrupt, which can be sent from one process to another, or from one thread to another.
When we terminate a process by typing 'kill –9' we are sending signal 9 to the process ID which follows this command. The shell passes the 'kill' command to the kernel, which logs the pending interrupt in the interrupt table of the process. When it becomes the turn of the process to run, the kernel checks the signal against the signal mask lodged with the process and, if the signal is permitted to be handled, performs the action defined in the mask. Unfortunately, signal 9 is not of this type, so the process is terminated.

Apart from terminating processes, the kernel uses signals to inform processes of interesting occurrences, like the expiry of a timer, or the termination of a child process, and to warn of forthcoming disasters, like power failures and disk overflows.

Although most signals have predetermined functions, two are undefined, and are reserved for use by users. Not surprisingly, they are called SIGUSR1 and SIGUSR2.

How do we send a signal to a process?

First, any code which uses interrupts, has to include the signal definition header.

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

This file is basically a cross-reference of the human-readable signal names, and their numbers, which are the only way the kernel recognises them.

The actual system call, which sends the signal, is the same as its command line counterpart. To terminate a process, we use:

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

Obviously, before this call can succeed, we must have permission to use it. For the examples which we shall consider, there is no problem, since our parent process owns all child processes but, if we wanted to send a signal to a process owned by another user, the question of user and group ID's would need to be taken into account.
The two arguments to kill(), are the process ID of our intended victim, and the number, or symbolic name of the signal we intend to send it.
Since we are mainly interested in the use of signals as a means of communication, rather than their use to slaughter innocent processes, we will be passing them between parent and child. The process ID's of all children are known to a parent, from the return value of fork(), and the process ID of a parent is returned by the getppid() system call.
For example, if we needed to tell our parent process that we had encountered a serious problem, we would use the following call:

```
If(kill(getppid(), SIGUSR1) == -1){
    perror(“Tell parent failed”);
 }
```
Note, that we use SIGUSR1, rather than ‘16’. Some of the numeric values assigned to signals (especially SIGUSR1 and SIGUSR2) vary, from one version of Unix to another so, in the interests of portability, we will only refer to signals by name.

Most operating systems have a version of the ‘kill()’ system call, which can be directed to a particular thread,, from within the process, in which the thread runs. Also, individual threads, which normally inherit their interrupt mask from the parent thread, which created them, can set up their own interrupt mask, which can be different to that of the parent process. When a signal is sent to a process, which has multiple threads running, it is delivered to the thread best able to accept it, so there is a very strong possibility, that the wrong combination of interrupt masks can lead to some interesting bugs, like thread deadlock.
Unless it is absolutely necessary, and all of the possibilities can be accounted for, it is a good idea to leave each thread with its default interrupt mask.

The most useful signals

The signal man page, in section 3head of the Unix manual, contains a complete list of all, thirty-odd, signals, with their definitions and numeric values. Here are a few important ones.

SIGCHLD

This is number 18 on Sun Solaris, and is sent by the kernel, to the parent process, each time a child process terminates. If we design our code carefully, we can keep track of our children, by counting the number of SIGCHLD signals received.

SIGALRM

The Unix kernel has an alarm clock, which can be used by a process, to wake itself up, put itself to sleep, or interrupt its normal processing, to perform some important task. The system call is alarm(int countdown) which sends the signal after countdown seconds.

SIGUSR1 and SIGUSR2

These can be anything we want them to be. Since there are enough predefined termination signals, we would probably use them to communicate events between processes, such as the state of a semaphore, the readiness of a process to send data or the presence of a file.

SIGSTOP

There may be occasions, when we want a process to stop execution, but be ready to run again. When SIGSTOP is received, the process, and each of its threads, stops executing. We can restart it again, by sending it a SIGCONT signal.
Alternatively, the process can be restarted, by sending it.a different signal, which is set to be caught in the interrupt handler. From the interrupt handler, we can either just execute return(), or call another function, or perform longjmp() to a fixed place in the code.
It should be mentioned, however, that if SIGSTOP or SIGCONT is sent to a child process, the kernel will send its parent a SIGCHLD signal, unless this is set to be ignored. Therefore, care should be exercised, in the use of these signals.

SIGHUP

This is the signal passed by ‘kill –1’ to a process. It may be considered as a hint that the process should die. It is intentionally left, able to be caught, so that a process can perform the last rites, before quitting. Frequently, this signal is used as a reconfiguration hint, forcing the process to re-read startup files, environment variables and other sources of configuration data.

SIGINT
When control-C is entered on the keyboard, it sends a ‘kill –2’ signal to the process attached to the window of which the keyboard is the standard input. The signal was designed to do, from the keyboard, what SIGHUP does from the command line.

SIGQUIT
When it is desired to obtain the core image of a process, for examination, or to measure how much memory the process is using, ‘kill –3’ will force the process to dump core. A ‘core dump’ (the term is a relic of the days when memory was a cubic foot of magnetic cores) is a dump, of the data segment, text segment and stack of a running process. Various fault conditions within the process will cause the kernel to send it this signal, but there are occasions, when we need to make a healthy process dump core. This signal cannot be caught.

SIGKILL
In the early Unix manuals, SIGKILL, or ‘kill –9’, was described as ‘kill with extreme prejudice’. It results in instant termination of the process, and cannot be caught.

SIGTERM
This signal has exactly the same effect as SIGKILL.

What is an interrupt mask?
SIGKILL, together with SIGTERM and SIGQUIT, cannot be caught and handled. If any of these appear in the interrupt table, the process is terminated. For all others, there is the interrupt mask.
An interrupt mask is a template of how the process proposes to deal with incoming interrupts, and is constructed using the signal() system call or, alternatively, the sigaction() function.
The signal() call has the following synopsis:

```
void (*signal(int signal_number, void (*handler)(int)))(int);
```

Which means, that the system call returns a pointer to void, and we pass it a signal number, (or its symbolic name), together with either:

- An int, usually represented symbolically, which defines the action.
- The address, of a function, which returns void, and takes an int as an argument.

Basically, there are only two useful actions that can be assigned to an interrupt mask. The incoming signal can be either ignored, or it can be caught, and handled by a special function. For example, if we want to ignore control-C, (which is SIGINT, or signal 2), we would code it like this:

```
signal(SIGINT, SIG_IGN);
```

Alternatively, if we wanted to exit gracefully, by calling a routine called cleanup(), to remove temporary files, before quitting, we would say:

```
signal(SIGINT, cleanup);
```

Where cleanup() was declared as

```
void cleanup(int);
```

The int argument to the interrupt handler is, in fact, the signal number, so that we can identify which interrupt we're handling.

The sequence of events, then, in designing a function to handle signals is as follows:

1. Create an interrupt mask, by defining the actions for all incoming signals, which are of interest. This is best done in a function, rather than in a block of inline code, since some versions of Unix will automatically reset the interrupt mask to its default state, after a signal has been received. This means that, the first time the signal arrives, everything works fine then, on all subsequent receipts of the same signal, nothing happens. The way to avoid many happy hours of staring at the debugger, is to call the signal mask routine, on exit from the interrupt handler. Our signal mask routine would be coded something like this:

```
int
setsignals()            /* setsignals */

{
   signal(SIGHUP,interrupt);     /* hangup, */
   signal(SIGINT,interrupt);     /* interrupt */
   signal(SIGQUIT,SIG_IGN);      /* quit */
   signal(SIGALRM,interrupt);    /* alarm clock timeout */
   signal(SIGCHLD,interrupt);    /* child stop or exit */
   signal(SIGILL,SIG_IGN);       /* illegal instruction (not reset when
caught)*/
   signal(SIGEMT,SIG_IGN);       /* EMT instruction */
   signal(SIGFPE,SIG_IGN);       /* floating point exception */
   signal(SIGSYS,SIG_IGN);       /* bad argument to system call */
   signal(SIGPIPE,SIG_IGN);      /* write on a pipe with no one to read it */
   signal(SIGTERM,SIG_IGN);      /* software termination signal */
   signal(SIGURG,interrupt);     /* urgent condition on socket or I/O channel
*/
   signal(SIGTSTP,SIG_IGN);      /* interactive stop */
   signal(SIGTTIN,SIG_IGN);      /* background read from control terminal */
   signal(SIGTTOU,SIG_IGN);      /* background write to control terminal */
   signal(SIGIO,SIG_IGN);        /* I/O possible, or completed */
   signal(SIGXFSZ,interrupt);    /* file size limit exceeded */
   signal(SIGWINCH,SIG_IGN); /* window size changed */
   signal(SIGPWR,interrupt);      /* power-fail restart */
   signal(SIGUSR1,SIG_IGN);      /* user defined signal 1 */
   signal(SIGUSR2,SIG_IGN);      /* user defined signal 2 */
   signal(SIGVTALRM,SIG_IGN); /* virtual time alarm */

}                    /* setsignals */
```

Note, that we have opted to ignore most of the signals, but we call our interrupt handler ('interrupt()') for SIGHUP, SIGINT, SIGALRM, SIGCHLD, SIGURG, SIGXFSZ and SIGPWR.

The three latter are emergency conditions in the machine, which might need some urgent action on our part, while SIGHUP and SIGINT need cleanup actions.
SIGALRM, on the other hand, is used by scheduling routines, while SIGCHLD is important, for keeping track of our child processes.

2. Create an interrupt handler.

An interrupt handler is basically just a decoder for the signal number, with an appropriate function call associated with he signals in which we are interested.
When an interrupt comes in, the kernel will place, on the stack, the address of the piece of code, which is currently executing, and then call the handler. On return from the handler, execution should resume from the point of interruption.
This may not always be desirable. If we are in the middle of a loop, or executing pause(), waiting for an interrupt to break out of the loop, or wake us up, then the last thing we want is to go back into the loop, or the pause() state.
In such cases, we can return from the handler, by way of longjmp(), instead of just falling through the bottom. A call to longjmp() should be used with care, however, since it zeros the stack, so there is no way of coming back from it.

A generic interrupt handler, which is the focal point for all of our signal activities, is shown in the next code example. It actually accommodates far more signals than the signal mask dictates, purely for exemplary purposes.

```
void
interrupt(sig)          /* interrupt */
int sig;
{
   printf("Received signal %d: Handling interrupt...\n", sig);

   switch(sig){
      case SIGPWR:         /* power failure imminent */
         printf("Alert: Power failure imminent. Take remedial action\n");
      break;
      case SIGXFSZ:        /* disk overflow */
         printf("Alert: Disk overflow imminent. Take remedial action\n");
      break;
      case SIGHUP:                  /* kill -1 */
         printf("Received SIGHUP. Updating environment...\n");
         update_environment();
         printf("Going back to wait for next interrupt\n");
         longjmp(there, 0);
      break;
      case SIGINT:                  /* ^C */
         printf("Ouch! Received SIGINT. Quitting\n");
         exit(0);
      break;
      case SIGILL:
         printf("Fatal: Illegal instruction\n");
      break;
      case SIGEMT:
         printf("Fatal: Illegal instruction\n");
      break;
      case SIGFPE:
         printf("Fatal: Floating point error\n");
      break;
      case SIGSYS:
         printf("Non-Fatal: mangled system call\n");
      break;
      case SIGPIPE:
         printf("Non-Fatal: Write on pipe with no reader\n");
      break;
      case SIGALRM:
          /* system alarm clock. We go and do something as a result, then
           * go to a special piece of code, rather than return to what
          */ we were doing
         special_routine();
         longjmp(there, 0);
      break;
      case SIGTERM:
         printf("Fatal: Killed with extreme prejudice\n");
      break;
      case SIGURG:
         printf("Fatal: I/O condition urgent\n");
      break;
      case SIGTSTP:
         printf("Non-Fatal: Process stopped by user\n");
      break;
      case SIGVTALRM:
         printf("Non-Fatal: Virtual alarm clock\n");
      break;
   }

   /* now reset the mask, since some versions of Unix trash the interrupt
table */

   setsignals();

}                  /* interrupt */
```

Note that the last action, before leaving the handler, is to reset the signal mask. The reason for this, is that some operating systems automatically reset it to default, after a signal has been received and handled.