Introduction to Using `sigaction`

17 March 2011

From `signal` to `sigaction`

Programs receive signals from various sources and for various causes. A user may press the interrupt or quit keys, the program may set an alarm, a user may resize the terminal window, the program may divide by zero or dereference a null pointer.

Programs are not at the mercy of these signals, though. A program can tell the kernel what to do when a signal arrives for the program. The program specifies one of the following: (a) take the default action, (b) ignore the signal, or (c) run a particular function. And the program may change these arrangements as often as it pleases.

How does a program set and change these arrangements? As is so often the case with Unix/Linux programming, the answer has two parts: how it used to be, and how it is now. In Unix/Linux, the past is never really dead , it just gets re-implemented as wrapper functions around the present.

How It Used to Be: `signal()`

Say a program wants to ignore the effect of the user pressing Ctrl-C, the key that generates the interrupt signal -- SIGINT. The program can call:

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

Alternatively if the program wants to execute a function called `wrapup` when it receives SIGINT, it can call:

```
signal( SIGINT , wrapup );
```

The function name is passed without any arguments even without parentheses. The function name is the address of a block of code, and the call to signal expects as arguments a signal and the address of the function to call.

Finally, if the program wants to revert to the default system handling of SIGINT, it calls:

```
signal( SIGINT , SIG_DFL );
```

Three simple choices, nothing more. Two arguments only -- the signal to handle and how to handle it. How to handle it is (a) to take the default action, (b) to take no action, or (c) to call a user-specified function. And that's it.

But What If I Don't Want Pickles and Mustard on My Burger?

Some people don't want pickles and mustard on their burgers. Some burger joints always put pickles and mustard on, some put on mustard but no pickles, some include pickles and extra ketchup. When you ask for a burger, you get the standard burger configuration for that restaurant.

That's how signal was on Unix in the old days. There were only three choices: default, ignore, call a function. The details of what happened when you called a function was up to the version of Unix. On some versions of Unix, you got pickles, and on some versions you didn't. And there was no way to have it your way. You just had to work around the standard behavior.

What Are the Pickles and Mustard of Signal Handling?

Handling signals turns out to be a complex topic. Many different conditions can occur. For example, imagine your program arranges to call a function called `wrapup` when it receives SIGINT. Sounds pretty simple. But say the user presses Ctrl-C twice. The first keypress generates a signal, and control jumps to the function. As the program is executing code in the function, a second SIGINT arrives. Now what?

There are at least three different possible responses. (a) One response to two signals in a row is to invoke the handler function a second time right in the middle of the first invocation of the handler function, and when done with the second invocation return to the first invocation. (b) Another response would be to queue the second signal, complete the first invocation of the function, then dequeue the second signal and invoke the handler functions again. (c) A third response would be to apply the default action (process termination) to the second appearance of the signal.

But a call to `signal()` does not allow the programmer to specify which of these responses to take. The programmer gets whatever response the version of Unix uses. And not every version of Unix uses the same response. In some Unix versions you get pickles, and in some Unix versions, you don't. And with the simple interface, you don't get to choose.

Have It Your Way

The original signal-handling model for Unix is simple and comprehensible. You have three choices -- default, ignore, handle -- and you have one easy-to-use function. But, as noted above, there are two main problems. First, you have no way to specify how you want the kernel to handle more complex combinations of signals (like two of them in a row). Second, not every version of Unix handles the more complex combinations the same way. So, even if you like the standard way Unix does it, you cannot count on that standard way being standard on other versions of Unix.

The solution to these two problems is to give the programmer complete control of signal-handling policy. Furthermore, this advanced solution allows the programmer to specify different policies for different signals. In the context of our mustard and pickles analogy, you now have full reign in the kitchen.

New Additions to the Signal-Handling Interface

The old signal-handling interface had just two arguments: the signal to handle and how to handle it. The new interface still has those two arguments. In addition the new interface has two new arguments. One argument tells the kernel what to do with other signals when it is handling the specified signal. The other new argument tells the kernel what to do in various situations. With this new interface function, `sigaction()`, you specify the signal, what to do with it, what to do with other signals, and how to respond to various conditions.

To pass this larger number of arguments to the kernel's signal-handling code, you store values in a struct, and you pass the address of that struct to the kernel.

The struct looks like:

```
struct sigaction
{
        void ( *sa_handler )(int);      /* same as arg in signal         */
        sigset_t sa_mask;               /* what to do with OTHER signals */
        int      sa_flags;              /* bits for other settings       */
}
```

The sa_handler member is exactly the same as the second argument to the original `signal()` call. It can be SIG_IGN, SIG_DFL, or the name of a function to call. Nothing new there.

The other two members, `sa_mask` and `sa_flags`, are new. These new members let you tell the kernel how to respond to more complex situations.

Blocking Other Signals: `sa_mask`

The first new member allows you to tell the kernel to block specific signals when it is handling the current signal. For example, if you want to call a function to handle the SIGINT signal but want to delay handling a timer tick until you are done, then add SIGALRM to the signal set specified by `sa_mask`. You can add as many signals as you like to this signal set. When the handler for the specified signal is called, any arrivals of those other signals are blocked until the current handler is done. By doing this, your signal handler can feel secure that it won't be interrupted by those other signals when it is doing its work.

What type of variable is this set of signals to block? This signal set could be implemented as a long int with one bit per signal. Instead, the signal set is an abstract type, and the techniques for manipulating that set are all provided as functions. No direct modification of the set is needed.

Here are the methods:

```
#include <signal.h>
int sigemptyset(sigset_t * sigmask);
int sigaddset(sigset_t * sigmask, const int signal_num);
int sigdelset(sigset_t * sigmask, const int signal_num);
int sigfillset(sigset_t * sigmask);
int sigismember(const sigset_t * sigmask, const int signal_num);
```

As the names suggest, these functions let you remove all elements from a set, add a signal to a set, remove a signal from a set, add all signals to a set, and see if a signal belongs to a set. Once you install in a signal set your choice of signals to block, that member is ready.

All you need to do now is to decide which flags to set in the sa_flags member.

Specifying Policy: `sa_flags`

We saw in an earlier section various policies the kernel could use to handle two consecutive instances of one signal. We said then that Unix could chose any policy it liked. With the addition of sigaction and its sa_flags member, the kernel lets you chose the policy.

This member is an int with individual bits representing specific policy choices. We examine a few of the flags here. To see the complete list, check a reference text or search the web for "sigaction sa_flags".

`SA_RESETHAND`

If this bit is set, the kernel will reset the action for this signal to its default value, usually process termination. This bit allows programs to emulate the original, one-shot model of signal handling. Unless you are porting a very old application that really needs that behavior, there is no no reason to use this.

`SA_RESTART`

Say your program handles timer ticks (SIGALRM) by calling some function. Say furthermore that a timer tick arrives when your program is waiting for input in a function such as getchar(). Signal handlers, we have said, work by interrupting the current code, jumping into the handler, and when done, returning to the point of interruption. For a concrete example, consider

```
printf("pickles (y/n)? ");   /* A: ask question */
x = getchar();               /* B: get reply    */
if ( x == 'y' )              /* C: act on reply */
```

Say a signal, like a timer tick, arrives during the wait at step B. The signal handler executes, then control returns to the previous step in the program. Does the program resume waiting *or* does the program stop waiting and return some indication it was interrupted?

It turns out there are two responses: (a) return to waiting for input or (b) don't return to waiting for input. In case (a), we say the slow system call is automatically restarted. In case (b), the input instruction returns with an error, errno is set to EINTR, and execution resumes at step C.

The default action on the version of Linux at the Science Center is response (a) -- restart. Some versions of Unix use response (b) -- return EOF with errno set to EINTR. If you want your program to use response (b) clear this bit. Note: specifying response (b) requires a bit more coding to check for a return value of -1 from read or EOF from getchar, but this choice gives your program more control of how it handles signals.

`SA_NODEFER`

When the program is handling a signal, another instance of that signal is blocked until the first instance is done being handled. This default behavior prevents the handler for one signal being

interrupted to run the same handler. The risk of one handler interrupting itself is the potential for corrupting data structures used by that function. If you prefer to allow one signal to interrupt its own hander, then set this bit. To protect against corrupted data, be sure the handler function is written to work even if called from within itself.

Translating `signal()` to `sigaction()`

Replacing a call to `signal()` into a call to `sigaction()` looks like:

before

```
void wrapup(int);                   /* declare hander */

signal( SIGINT, wrapup );           /* attach handler to signal */
```

after

```
void wrapup(int);                   /* declare handler */
struct sigaction new_act;           /* create two structs */
struct sigaction old_act;

new_act.sa_handler = wrapup;        /* set handler */
sigemptyset( &new_act.sa_mask );    /* no signals to block while handling */
new_act.sa_flags = 0;               /* no special policy choices */
sigaction( SIGINT, &new_act, &old_act );
```

If you do not want to record the previous settings for that signal, then send a null pointer as the argument for old_act. Of course, check return values from the system calls to see if anything went wrong.

What about Blocking Signals at Other Times?

When two instances of a signal arrive in rapid succession, the second instance is blocked until the first instance is handled. If you prefer to have the second instance interrupt, rather than follow, the first instance, you can set the SA_NODEFER bit in the flag set.

If a different signal arrives when one signal is being handled, the other signal will be handled unless that signal is included in the signal set sa_mask. If a signal is included in the sa_mask signal set, that signal will be blocked until the current signal is done being handled. But what if you want to block specific signals even when your program is not in a signal handler?

For example, a function might be modifying a data structure that is used by signal handlers. If that operation was interrupted when the data structure is in transition, for example a linked list with some pointers not yet moved, then the signal handler would find inconsistent, incomplete, or corrupt data. This is an example of a good time to block some signals. The system function that allows you to block and unblock signals at any time is called `sigprocmask()` . There are many textbook and online references that explain how to use this function.

Wrapping Up

Signals are the basis of effective, efficient multi-tasking. Just as ringtones on a telephone, knocks on the door, alarms from your wristwatch, beeping horns from other motorists are signals that allow you to direct your attention to tasks needing action, SIGINT, SIGALRM, SIGWINCH, and SIGFPE are signals from users, timers, window managers, and computation units that invite your program to direct its attention to tasks needing action. Deciding how to respond to those signals, which to handle when, which to defer, which to ignore, etc is a complex task. Subtle errors in planning can lead to drastic problems.

Unix/Linux began with a simple model for handling signals. You could ignore a signal, be killed by a signal, or handle a signal by calling a function. Soon, that simplicity turned out to be inadequate for a variety of complex multi-tasking needs. Therefore, a more powerful mechanism was designed -- sigaction and sigprocmask. These two tools give programmers much finer and more flexible control over how a program manages and responds to signals.