by Peter Collinson, **Hillside Systems**

WENDY GROSSMAN

Signals

One of the many aspects of Mr. Gates' products that niggle me is the inability to stop something from happening. You start an application doing something by mistake and find yourself pounding random keys to stop it. Will this stop with the Escape key? Should I hit Break? Should I use Control-C? Should I use Control-Alt-Delete? At least when you use the latter key chord on Windows NT you have the ability to kill the offending process using the Task Manager, always hoping that the application resources are recovered properly by the system. Starting activities erroneously happens all too often. The problem seems to trouble me more these days, because age has caused me to be marginally optically challenged and I get one-off errors on menus with little apparent effort.

I suppose that the inability to stop things from happening occurs on UNIX too, but it's much more rare and is usually a feature of a system failure or a badly programmed application. UNIX was designed to allow the keyboard user to send an emergency message to an application and for that application to be stopped in its tracks. Control is then returned to the user, where it should be. The emergency message is part of a system of simple interprocess communications (IPC) called *signals*.

A UNIX system consists of a memory-resident kernel and many processes that do all the work. Processes are programmed using a model where each process exists alone, running in a virtual machine. A process has access to an address space and talks to the outside world via system calls that switch into the kernel. A process cannot directly see other processes running on the machine and cannot interfere with the address space of other processes unless special arrangements are made to share memory. Processes talk to the kernel mostly using data streams. I discussed stream I/O concepts in my column entitled "Device Independence" (July 1998, Page 28). Sending data from one process to another doesn't happen directly, the data is passed into the kernel from the sending process and is read from the kernel by the receiving process.

This isolationist model has many positive aspects. It means that if a process fails for one reason or another, its misbehavior cannot randomly affect another process, possibly causing unexpected events or strange unprovable data corruptions. The action of each process is deterministic, "garbage in" will mean "garbage out," but it will be the same garbage every time. The safety that the process environment supplies is particularly important for the C language, where errors in coding can easily cause a program to spray data over its address space. The process model limits the damage.

Signals are part of that model. A signal is not actually a message. It's a bit in a word in the control block for the process that's maintained by the kernel. When the kernel wishes to post a signal to a process, it simply sets the appropriate bit. When the kernel next decides to run the process, it looks at the set of signal bits and, if any are set, will start the process running using some special code to handle the signal.

Well, you may have been confused by “when the kernel next decides to run the process” and you have a right to be, surely a process is running all the time? On a processor with a single CPU, a process will run until it makes a system call, which will be serviced by switching into the kernel. If the system call takes some time, perhaps reading some data from a disk, the process is generally put to sleep until the data is ready. When the data materializes, the process is ready to go and can be placed in the run queue, where it is allowed to have some more CPU cycles. Of course, the process isn’t aware of this sleeping time, as far as it’s concerned it has made a system call and had some data returned.

You will find that core files materialize on all UNIX systems from time to time and, in general, they signal that something has failed and can be deleted.

The hardware may also cause a switch into kernel mode. The hardware will interrupt the CPU when a device completes some physical action, causing a switch into the kernel to service the peripheral. One device that’s always present is a clock that ticks away at regular intervals allowing the kernel to share the CPU more fairly between the running processes. The kernel can see that one process has managed to get some computation done and now it’s the turn of another process to run. So, the system is always switching in and out of kernel mode, deciding what process is to be run next and starting that process.

Finally, just before a process is resumed at the point where it left off, the kernel checks the signal bits and calls a special piece of code to handle the signal rather than restarting the process as if nothing had happened.

Default Signal Actions

The signal-handling code has a set of default actions that are performed depending on the type of signal. For most signals, the process can also elect to provide its own signal-handling code and take whatever action the programmer sees fit. However, most processes don’t take any special action and just follow the standard default rules.

The first, and most common, default action makes the process exit immediately. The signal-handling code will call the `exit` system call and the process will finish. Note that the process exits, it’s not really terminated by the system. The `exit` system call takes a value and this is passed to the process that started the command. By convention, on a normal exit, a zero value is used to mean the successful termination of a program. The signal code will set some bits in this returned value, so the calling process can see that its child has died because it received a signal. Of course, some programs don’t care about this information. Some just inform the user. For example, shells print out a message saying the process has died because of a signal. Some processes depend on knowing what has happened to one of their children and will take special action when a nonzero value is returned.

The second default signal action causes the execution state of the program to be dumped into a file called `core` and then the process will exit as above. Of course, memory hasn’t been “magnetic core” for some considerable time, so the name of this file betrays its antiquity. Recent releases of 4.4BSD have tended to use a name that relates to the original command, so if the dump is from a program called `fiona`, the core dump will be in a file named `fiona.core`. Using a command-specific name helps get around the problem of `core` files being overwritten by several crashes of different processes.

The execution state of the program can be useful if you’re debugging a program or want to attempt to find out what it was doing before it was sent the signal. Debugging programs can use the core dump to investigate the state of the program’s address space. Core dumps can be large, programs often have immense address spaces that rely on the virtual memory capabilities of the operating system and the host computer. To overcome the problem of filling up disks with immense core files, most recent versions of UNIX have limits that prevent the file being written if it will be larger than a predefined size.

Core dumps also present a security hazard. If I can get a core dump from the `login` program immediately after you have typed your password, then I can see your password in plain text (if I know how to use an appropriate debugger). Again, this problem is fixed on most current UNIX systems. One step is to prohibit anyone from obtaining a core dump from a program that has the `setuid` bit set in the file where the command lives in the file system.

You will find that `core` files materialize on all UNIX systems from time to time and, in general, they signal that something has failed and can be deleted. Although, if you have the knowledge, it may be worth your while trying to deduce what provoked the dump. On Solaris, the `file` command will tell you the name of the program that was responsible for the core, which can be useful.

The third default action for a signal is actually an inaction, the signal is ignored. In reality, if the programmer hasn’t set up a special signal-handling routine, then an ignored signal is simply not posted. The fourth and final system action is to suspend the process. Most of the documentation calls this action “stopping” the process. I tend to resist using this term because it implies finality. The process is only halted temporarily and “suspending” is a more accurate term.

Using Signals

As I said, the programmer can specify that a routine in the process should be called when the signal occurs, rather than taking the default action. There are a number of reasons why this is desirable, why a program may wish to trap signals. What follows examines some possibilities and is by no means an exhaustive list.

First, a common case: If a program creates a temporary file, it’s nice to be able to clean up when the user wants it to stop. Thus, the code will catch the signal, delete the file and exit. The program is doing what the user wants–stopping, but being tidy too.

Second, if a program is interactive, an editor, for example, you may wish to use the signal to stop the current command in the editor rather than causing the whole program to exit, which will lose the file you are editing. Again, the program is doing what is natural: The shell is an interactive program and doesn't die when the user emits a signal from the keyboard. The interactive program is mimicking that behavior.

Third, there are several background processes that trap the "hang-up" signal and on its receipt will call a routine that rereads their configuration file. The signal is acting as a restart facility and is an example of a simple piece of interprocess communications.

As a final example, there is a class of programs to which you might want to send a message. A good example is a line printer spooler, which will sit waiting for files to appear in its spooling directory so that it may print them. It's not very efficient to make such a program continuously look for work. It's more productive to send it a message that says, "there is work now," and this message can be a signal. Actually, these days, many such programs are programmed with sockets so that a datagram is sent to tell it to spring into life and print something.

Catching Signals

To catch a signal, the programmer specifies that one of the routines in a program is to be called when the signal occurs. A call to the handling code is seen as an "unexpected" routine call to the program. In modern terms, signals create simple multithreading in the program, where there are two distinct paths in the code sharing the same data space.

In the original UNIX systems, care had to be taken in the data space of an application that was to continue to run after a signal had been caught. It was possible to have races, problems with shared data and all the evils of multithreading, but there was no support for mutual exclusion and locking that we expect in multithreaded environments today.

The basic design of signal handling was somewhat flawed because it was not really envisaged that signals would be used as an IPC mechanism. When the signal-handling routine was called, the kernel reset the action for that signal back to the default. Usually, the first action in the handling routine was to reestablish the handling routine. However, this left a small window of opportunity where a signal could arrive, the routine was called, the default action was now in force and some code was needed to be executed to set up the handler. Another signal that arrived before the handler was reestablished could cause the process to take the default action, rather than calling the catching routine.

I recall spending an enormous amount of time wondering why my line printer spooler would occasionally crash, creating a core dump. At the time, UNIX didn't have a spooler and so I had written one. I never got to the bottom of the problem, and it was years later when I realized the signals were being set back to the default and the default for the signal I was using created a core dump.

The problems with unreliable delivery of signals caused the team at the University of California to develop many solutions that appeared in successive BSD releases. The final model that exists in most modern UNIX systems treats signals like hardware interrupts to the virtual machine that is the process. When the kernel calls the signal-handling routine for a particular signal, that signal is *masked*. If the signal arrives while the handler code is being called, then it will be posted, but the catcher routine will not be called until its first call exits or takes action to allow the signal. The 4.4BSD signal model was adopted by POSIX (with some changes) and forms the basis of signal handling today.

Sending Signals

The standard terminal handler allows the user to send three signals to all the processes attached to the terminal. You can bind these signals to different keys using the `stty` command. The first, the interrupt signal (`SIGINT`), is usually bound to Control-C on most UNIX systems. The default action for the interrupt signal causes the running processes to exit. The second, the quit signal (`SIGQUIT`), is usually bound to Control-\ and causes a core dump and an exit. The quit signal is of more use to programmers and software developers than mortals. The third signal (`SIGTSTP`) is usually bound to Control-Z and delivers the suspend signal, stopping the processes from running. The terminal interface can also automatically deliver a signal telling all its processes to die when a carrier drops, which happens when a phone line is disconnected. By default, a process will exit when it receives the `SIGHUP` signal. This signal gave rise to a UNIX command, `nohup`, which starts a user-defined command that ignores the `SIGHUP` signal. This was originally used by people who wanted to dial in, start a background job and then disconnect the phone line, leaving the background job running.

The `kill` command is used to send a signal to one or more processes. By default, if you simply supply it with a process ID, it will send the terminate signal (`SIGTERM`) to the appropriate process. The default action for this signal causes the process to exit. However, `SIGTERM` can be caught and ignored.

To kill a process dead, you need to send the kill signal (`SIGKILL`). This signal cannot be caught or ignored and is guaranteed to kill the process (assuming that it's not waiting on some system event that will never happen). You can deliver the mortal blow by saying

```
$ kill -s SIGKILL <pid>
```

but this is too much typing. I prefer the older form

```
$ kill -9 <pid>
```

which sends signal number 9, which happens to be `SIGKILL` on my system (and probably on yours too, it's one of those immutable constants that we know and love). You do need to be circumspect in your use of `SIGKILL`; if the process maintains temporary files, then it may expect to clean up and will do so when it is sent a `SIGTERM`. The usual strategy is to send a `SIGTERM` and, if that fails, send a `SIGKILL`.

As I said above, several daemon processes are coded to use the `SIGHUP` signal to reload tables. Many modern versions of such programs create a `.pid` file in some known spot on the file system and provide a shell script that "restarts" the daemon. The shell script reads the `.pid` file and sends the `SIGHUP` signal. Using `ps` to find the process ID and typing

```
$ kill -1 <pid>
```

works too.

Sending Signals to the Right Process

Of course, you hit Control-C on the keyboard without a thought and see the current running foreground process exit because it received a `SIGINT` signal. Actually, there is a sophisticated mechanism ensuring that only the "right" process (or processes) receives the signal.

The first UNIX systems used the notion of the "controlling terminal" to decide which processes should receive a signal from the keyboard. All the processes that are talking to the terminal will be sent the signal. This seems reasonable, but let's go through things slowly and see what the implications are.

When you login to a historical UNIX system, your shell is connected to your terminal. There's only one process and it will ignore the interrupt signal because shells cling to life and don't want to die when you type Control-C. Now you type a command name into the shell and start another process. You think of this as the "foreground" process because the shell goes to sleep until the command exits. In reality, you have two processes running on the machine with equal status, but one is waiting for the other to die. Typing Control-C will cause a signal to be sent to both processes. The shell is still ignoring it and the "foreground" process will exit, assuming it has taken no special action to handle the signal. The shell is woken up because the foreground process has died and you can type a new command.

Next, you start a command sequence that contains several processes connected by pipes. Again, when you type Control-C, all the processes are sent the signal. The pipeline processes die and the shell wakes up. This does the "right" thing.

What happens when you start the pipeline but add an ampersand at the end of the line making a "background" process? When you type Control-C, the signal gets sent to all the processes and will kill your background pipeline sequence as well. However, killing the background commands is not "correct" behavior. When we place something in the background, we don't want it to die when we type Control-C, the keystroke is supposed to be killing only the foreground process.

We need to program our shell so that when it starts a background process, that process will ignore the `SIGINT` signal. If the background job consists of several processes, then all the processes need to ignore the interrupt signal. So, now when Control-C is typed, processes in the "background" (and the shell) are ignoring the signal and remain running. All other processes see the signal and will die. Again, we seem to have what is the correct intuitive behavior.

But, how do we kill our background processes? The only way is to use the `ps` command to discover their process IDs and then use the `kill` command to send `SIGTERM` or `SIGKILL`.

Well, the above state of affairs existed for some time on UNIX until job control was implemented in the early '80s. Because job control was a BSD notion, and by then "System V was considered a standard," the mechanism was not really picked up in the System V world until quite recently.

Job control works by implementing the idea of a "process group." When the shell starts a command or a command sequence, that job is placed in a distinct process group. A single number is retained in all the constituent processes and the kernel is able to use that number to identify members of the group. The shell is able to manipulate which process group has control of the terminal by loading the group number into the terminal interface with a special system call.

Now when you type Control-C, the `SIGINT` signal is only sent to the processes in the process group currently loaded into that terminal. As a result, the shell can define foreground processes (those in the process group that is loaded into terminal) and background processes (all other process groups). Background processes are not allowed to read from the terminal, they are put to sleep if they attempt to do so. You can also optionally make background processes wait politely to output to the terminal should this be desirable. Job control defined a new standard signal, usually bound to Control-Z, which temporarily suspended a running process. Using a few keystrokes, the user had the ability to control which processes were in the foreground and which were in the background.

Job control was a big leap forward, it allowed users on terminal lines to multiplex several tasks on the same screen. It's argued that it is of limited use today where we have the ability to create several virtual terminals on our workstations. However, I still use job control because it's fast and it's easier to type Control-Z than reach for the mouse and open a new window.

Further Reading

There's loads of information on how signals are implemented in 4.4BSD in *The Design and Implementation of the 4.4BSD Operating System*, by Marshall Kirk McKusick, Keith Bostic, Michael J. Karels and John S. Quarterman, published by Addison-Wesley Publishing Co., 1996, ISBN 0-201-54979-4.

***Peter Collinson** runs his own UNIX consultancy, dedicated to earning enough money to allow him to pursue his own interests: doing whatever, whenever, wherever... He writes, teaches, consults and programs using Solaris running on a SPARCstation 2. Email:* `pc@cpg.com`.