# Sending and Trapping Signals

Signals are a basic tool for asynchronous interprocess communication. What that means is one process (A) can tell another process (B) to do something, at a time chosen by process A rather than process B. (Compare to process B looking for a file every few seconds; this is called polling, and the timing is controlled by process B, rather than process A.)

The operating system provides a finite number of signals which can be "sent" to tell a process to do something. The signals do not carry any additional information; the only information the process gets is which signal was received. The process does not even know who sent the signal.

Unless a process takes special action in advance, most signals are fatal; that is, the default action a process will perform upon receiving a signal is an immediate exit. (Exceptions: SIGCHLD is ignored by default, SIGSTOP pauses the process, and SIGCONT resumes the process.) Some signals (such as SIGQUIT) also cause a process to leave a core file, in addition to exiting.

# 1. Traps, or Signal Handlers

A process may choose to perform a different action, rather than exiting, upon receiving a signal. This is done by setting up a *signal handler* (or *trap*). The trap must be set before the signal is received. A process that receives a signal for which it has set a trap is said to have *caught* the signal.

The simplest signal handling a process can choose to perform is to *ignore* a signal. This is generally **a bad idea**, unless it is done for a very specific purpose. Ignoring signals often leads to *runaway processes* which consume all available CPU.

More commonly, traps can be set up to intercept a fatal signal, perform cleanup, and then exit gracefully. For example, a program that creates temporary files might wish to remove them before exiting. If the program is forced to exit by a signal, it won't be able to remove the files unless it catches the signal.

In a shell script, the command to set up a signal handler is `trap`. The `trap` command has 5 different ways to be used:

- `trap 'some code' signal list` -- using this form, a signal handler is set up for each signal in the list. When one of these signals is received, the commands in the first argument will be executed.
- `trap '' signal list` -- using this form, each signal in the list will be ignored. Most scripts should not do this.
- `trap - signal list` -- using this form, each signal in the list will be restored to its default behavior.

- `trap signal` -- using this form, the one signal listed will be restored to its default behavior. (This is legacy syntax.)
- `trap` -- with no arguments, print a list of signal handlers.

Signals may be specified using a number, or using a symbolic name. The symbolic name is greatly preferred for POSIX or Bash scripts, because the mapping from signal numbers to actual signals can vary slightly across operating systems. For Bourne shells, the numbers may be required.

There is a core set of signals common to all Unix-like operating systems whose numbers realistically never change; the most common of these are:

| Name | Number | Meaning |
| --- | --- | --- |
| HUP | 1 | Hang Up. The controlling terminal has gone away. |
| INT | 2 | Interrupt. The user has pressed the interrupt key (usually Ctrl-C or DEL). |
| QUIT | 3 | Quit. The user has pressed the quit key (usually Ctrl-\). Exit and dump core. |
| KILL | 9 | Kill. This signal cannot be caught or ignored. Unconditionally fatal. No cleanup possible. |
| TERM | 15 | Terminate. This is the default signal sent by the `kill` command. |
| EXIT | 0 | Not really a signal. In a bash or ksh (some implementations only) script, an EXIT trap is run on any exit, signalled or not. In other POSIX shells only when the shell process exits. |

Bash accepts signal names with or without a leading SIG for both the `trap` and `kill` builtin commands unless it is in POSIX mode when `kill` does not accept a leading SIG but `trap` does.

Thus the leading SIG is never required so SIGHUP can always be trapped by using `trap ... HUP` and sent by using `kill -s HUP process_ID`.

The special name EXIT is defined by POSIX and is preferred for any signal handler that simply wants to clean up upon exiting, rather than doing anything complex. Using 0 instead of EXIT is also allowed in a `trap` command (but 0 is not a valid signal number, and `kill -s 0` has a completely different meaning). So to clean up, just trap EXIT and call a cleanup function from there. Don't trap a bunch of signals. Sadly, this only seems to work in Bash. Other shells, such as zsh or dash, trigger the EXIT trap only if no other signal caused the exit. One further caveat: This only works in non-interactive shells (scripts), EXIT is not called if interactive shells get signalled (possibly a bug).

If you are asking a program to terminate, you should always use SIGTERM (simply `kill process_ID`). This will give the program a chance to catch the signal and clean up. If you use SIGKILL, the program cannot clean up, and may leave files in a corrupted state.

Please see ProcessManagement for a more thorough explanation of how processes work and interact.

# 2. Examples

This is the basic method for setting up a trap to clean up temporary files:

Toggle line numbers

```bash
#!/bin/bash
tempfile=$(mktemp) || exit
trap 'rm -f "$tempfile"' EXIT
...
```

This example defines a signal handler that will re-read a configuration file on SIGHUP. This is a common technique used by long-running daemon processes, so that they do not need to be restarted from scratch when a configuration variable is changed.

```bash
Toggle line numbers

#!/bin/bash
config=/etc/myscript/config
read_config() { test -r "$config" && . "$config"; }
read_config
trap 'read_config' HUP
while true; do
   ...
```

Setting a trap overwrites a previous trap on a given signal. There's no direct way to access the command associated with a trap as a string in order to save and restore the state of a trap. However, `trap` creates properly escaped output that's safe for reuse. The POSIX-recommended method is:

```bash
traps="$(trap)"
...
eval "$traps"
```

# 3. When is the signal handled?

When bash is executing an external command in the foreground, it does not handle any signals received until the foreground process terminates. This is important when you have a script like this:

```bash
trap 'echo "doing some cleaning"' EXIT
echo waiting a bit
sleep 10000
```

If you kill the script (using `kill pid` from another terminal, not with Ctrl-C -- more on that later), bash will wait for `sleep` to exit before calling the trap. That's probably not what you expect. A workaround is to use a builtin that will be interrupted, such as `wait`:

```bash
trap 'echo "doing some cleaning"' EXIT
echo waiting a bit
sleep 10000 & wait $!
```

Note that `sleep 10000` will not be killed and will continue to run. If you want the background job to be killed when the script it killed, add that to the trap. This kind of cleanup is *precisely* what traps are for!

```bash
pid=
trap '[[ $pid ]] && kill "$pid"' EXIT
sleep 10000 & pid=$!
wait
pid=
```

Any bash internal command will be interrupted by a (non-ignored) incoming signal. If you don't like `wait`, you can create an "infinite sleep" by attempting to open for reading a named pipe that will never have any writer. You can use any builtin as it will never be run anyway and the command will appear to block forever, without needing an external `sleep` to keep track of:

```
trap 'echo "we get signal"; rm -f ~/fifo' EXIT
mkfifo -m 0400 ~/fifo || exit
echo "sleeping"
true < ~/fifo
```

# 4. What is Ctrl-C doing?

You might think the first example in the previous section is not correct because when you try the script in your terminal and press Ctrl-C the message is clearly printed immediately. The difference between sending INT using `kill -s INT pid` and Ctrl-C is that Ctrl-C sends INT to the **process group**, which means that `sleep` will also receive the signal, and not just the script.

To send INT to a process group you need to use the process group ID preceded by a dash:

```
kill -s INT -123 # will kill the process group with the ID 123
```

To find the process group ID of a process you can use: `ps -p "$pid" -o pgid` (assuming your `ps` has this syntax). Note that you can't rely on the process group id of a script to be the same as `$$`, as that depends greatly on how the script was started. Also note that there may very well be other unrelated processes in the script's process group like when the script was started from another script or as part of a complex command line.

# 5. Special Note On SIGINT and SIGQUIT

If you choose to set up a handler for SIGINT (rather than using the EXIT trap), you should be aware that a process that exits in response to SIGINT 🌐 should kill itself with SIGINT rather than simply exiting, to avoid causing problems for its caller. The same goes for SIGQUIT. Thus:

```
trap 'rm -f "$tempfile"; trap - INT; kill -s INT "$$"' INT
```

bash is among a few shells that implement a *wait and cooperative exit* approach at handling SIGINT/SIGQUIT delivery. When interpreting a script, upon receiving a SIGINT, it doesn't exit straight away but instead waits for the currently running command to return and only exits (by killing itself with SIGINT) if that command was also killed by that SIGINT. The idea is that if your script calls `vi` for instance, and you press Ctrl+C within `vi` to cancel an action, that should not be considered as a request to abort the script.

So imagine you're writing a script and that script exits normally upon receiving SIGINT. That means that if that script is invoked from another bash script, Ctrl-C will no longer interrupt that other script.

This kind of problem can be seen with actual commands that do exit normally upon SIGINT by design. `ping host` returns with `0` when `host` is reachable (the ping has been answered) and non-zero otherwise when interrupted (which is the only way for `ping` to return in that case.

But in:

```
for i in {1..254}; do
  ping -c 2 "192.168.1.$i"
done
```

Here, if the user presses Ctrl-C during the loop, it will terminate the current `ping` command, but it will *not* terminate the loop. As Ctrl-C will just terminate the current `ping` invocation which will return with either 0 or 1 and `bash` will assume it was not interrupted and will not exit itself.

Commands that don't have a SIGINT handler (like `sleep`) or do the right thing of killing themselves with `SIGINT` upon receiving `SIGINT` (like `bash` itself does) don't exhibit the problem.

```
i=1
while [ "$i" -le 100 ]; do
  printf "%d " "$i"
  i=$((i+1))
  sleep 1
done
echo
```

If we press Ctrl-C during this loop, `sleep` will receive the SIGINT and die from it (`sleep` does not catch SIGINT). The shell sees that the `sleep` died from SIGINT. In the case of an interactive shell, this terminates the loop. In the case of a script, the whole script will exit, unless the script itself traps SIGINT.

CategoryShell