Package signal go1.15.2 Latest

Published: **Sep 9, 2020** | License: BSD-3-Clause | Standard library

Overview

Package signal implements access to incoming signals.

Signals are primarily used on Unix-like systems. For the use of this package on Windows and Plan 9, see below.

Types of signals

The signals SIGKILL and SIGSTOP may not be caught by a program, and therefore cannot be affected by this package.

Synchronous signals are signals triggered by errors in program execution: SIGBUS, SIGFPE, and SIGSEGV. These are only considered synchronous when caused by program execution, not when sent using os.Process.Kill or the kill program or some similar mechanism. In general, except as discussed below, Go programs will convert a synchronous signal into a run-time panic.

The remaining signals are asynchronous signals. They are not triggered by program errors, but are instead sent from the kernel or from some other program.

Of the asynchronous signals, the SIGHUP signal is sent when a program loses its controlling terminal. The SIGINT signal is sent when the user at the controlling terminal presses the interrupt character, which by default is ^C (Control-C). The SIGQUIT signal is sent when the user at the controlling terminal presses the quit character, which by default is ^\ (Control-Backslash). In general you can cause a program to simply exit by pressing ^C, and you can cause it to exit with a stack dump by pressing ^\.

Default behavior of signals in Go programs

By default, a synchronous signal is converted into a run-time panic. A SIGHUP, SIGINT, or SIGTERM signal causes the program to exit. A SIGQUIT, SIGILL, SIGTRAP, SIGABRT, SIGSTKFLT, SIGEMT, or SIGSYS signal causes the program to exit with a stack dump. A SIGTSTP, SIGTTIN, or SIGTTOU signal gets the system default behavior (these signals are used by the shell for job control). The SIGPROF signal is handled directly by the Go runtime to implement runtime.CPUProfile. Other signals will be caught but no action will be taken.

If the Go program is started with either SIGHUP or SIGINT ignored (signal handler set to SIG_IGN), they will remain ignored.

If the Go program is started with a non-empty signal mask, that will generally be honored. However, some signals are explicitly unblocked: the synchronous signals, SIGILL, SIGTRAP, SIGSTKFLT, SIGCHLD, SIGPROF, and, on GNU/Linux, signals 32 (SIGCANCEL) and 33 (SIGSETXID) (SIGCANCEL and SIGSETXID are used internally by glibc). Subprocesses started by os.Exec, or by the os/exec package, will inherit the modified signal mask.

Changing the behavior of signals in Go programs

The functions in this package allow a program to change the way Go programs handle signals.

Notify disables the default behavior for a given set of asynchronous signals and instead delivers them over one or more registered channels. Specifically, it applies to the signals SIGHUP, SIGINT, SIGQUIT, SIGABRT, and SIGTERM. It also applies to the job control signals SIGTSTP, SIGTTIN, and SIGTTOU, in which case the system default behavior does not occur. It also applies to some signals that otherwise cause no action: SIGUSR1, SIGUSR2, SIGPIPE, SIGALRM, SIGCHLD, SIGCONT, SIGURG, SIGXCPU, SIGXFSZ, SIGVTALRM, SIGWINCH, SIGIO, SIGPWR, SIGSYS, SIGINFO, SIGTHR, SIGWAITING, SIGLWP, SIGFREEZE, SIGTHAW, SIGLOST, SIGXRES, SIGJVM1, SIGJVM2, and any real time signals used on the system. Note that not all of these signals are available on all systems.

If the program was started with SIGHUP or SIGINT ignored, and Notify is called for either signal, a signal handler will be installed for that signal and it will no longer be ignored. If, later, Reset or Ignore is called for that signal, or Stop is called on all channels passed to Notify for that signal, the signal will once again be ignored. Reset will restore the system default behavior for the signal, while Ignore will cause the system to ignore the signal entirely.

If the program is started with a non-empty signal mask, some signals will be explicitly unblocked as described above. If Notify is called for a blocked signal, it will be unblocked. If, later, Reset is called for that signal, or Stop is called on all channels passed to Notify for that signal, the signal will once again be blocked.

SIGPIPE

When a Go program writes to a broken pipe, the kernel will raise a SIGPIPE signal.

If the program has not called Notify to receive SIGPIPE signals, then the behavior depends on the file descriptor number. A write to a broken pipe on file descriptors 1 or 2 (standard output or standard error) will cause the program to exit with a SIGPIPE signal. A write to a broken pipe on some other file descriptor will take no action on the SIGPIPE signal, and the write will fail with an EPIPE error.

If the program has called Notify to receive SIGPIPE signals, the file descriptor number does not matter. The SIGPIPE signal will be delivered to the Notify channel, and the write will fail with an EPIPE error.

This means that, by default, command line programs will behave like typical Unix command line programs, while other programs will not crash with SIGPIPE when writing to a closed network connection.

Go programs that use cgo or SWIG

In a Go program that includes non-Go code, typically C/C++ code accessed using cgo or SWIG, Go's startup code normally runs first. It configures the signal handlers as expected by the Go runtime, before the non-Go startup code runs. If the non-Go startup code wishes to install its own signal handlers, it must take certain steps to keep Go working well. This section documents those steps and the overall effect changes to signal handler settings by the non-Go code can have on Go programs. In rare cases, the non-Go code may run before the Go code, in which case the next section also applies.

If the non-Go code called by the Go program does not change any signal handlers or masks, then the behavior is the same as for a pure Go program.

If the non-Go code installs any signal handlers, it must use the SA_ONSTACK flag with sigaction. Failing to do so is likely to cause the program to crash if the signal is received. Go programs routinely run with a limited stack, and therefore set up an alternate signal stack. Also, the Go standard library expects that any signal handlers will use the SA_RESTART flag. Failing to do so may cause some library calls to return "interrupted system call" errors.

If the non-Go code installs a signal handler for any of the synchronous signals (SIGBUS, SIGFPE, SIGSEGV), then it should record the existing Go signal handler. If those signals occur while executing Go code, it should invoke the Go signal handler (whether the signal occurs while executing Go code can be determined by looking at the PC passed to the signal handler). Otherwise some Go run-time panics will not occur as expected.

If the non-Go code installs a signal handler for any of the asynchronous signals, it may invoke the Go signal handler or not as it chooses. Naturally, if it does not invoke the Go signal handler, the Go behavior described above will not occur. This can be an issue with the SIGPROF signal in particular.

The non-Go code should not change the signal mask on any threads created by the Go runtime. If the non-Go code starts new threads of its own, it may set the signal mask as it pleases.

If the non-Go code starts a new thread, changes the signal mask, and then invokes a Go function in that thread, the Go runtime will automatically unblock certain signals: the synchronous signals, SIGILL, SIGTRAP, SIGSTKFLT, SIGCHLD, SIGPROF, SIGCANCEL, and SIGSETXID. When the Go function returns, the non-Go signal mask will be restored.

If the Go signal handler is invoked on a non-Go thread not running Go code, the handler generally forwards the signal to the non-Go code, as follows. If the signal is SIGPROF, the Go handler does nothing. Otherwise, the Go handler removes itself, unblocks the signal, and raises it again, to invoke any non-Go handler or default system handler. If the program does not exit, the Go handler then reinstalls itself and continues execution of the program.

Non-Go programs that call Go code

When Go code is built with options like -buildmode=c-shared, it will be run as part of an existing non-Go program. The non-Go code may have already installed signal handlers when the Go code starts (that may also happen in unusual cases when using cgo or SWIG; in that case, the discussion here applies). For -buildmode=c-archive the Go runtime will initialize signals at global constructor time. For -buildmode=c-shared the Go runtime will initialize signals when the shared library is loaded.

If the Go runtime sees an existing signal handler for the SIGCANCEL or SIGSETXID signals (which are used only on GNU/Linux), it will turn on the SA_ONSTACK flag and otherwise keep the signal handler.

For the synchronous signals and SIGPIPE, the Go runtime will install a signal handler. It will save any existing signal handler. If a synchronous signal arrives while executing non-Go code, the Go runtime will invoke the existing signal handler instead of the Go signal handler.

Go code built with -buildmode=c-archive or -buildmode=c-shared will not install any other signal handlers by default. If there is an existing signal handler, the Go runtime will turn on the SA_ONSTACK flag and otherwise keep the signal handler. If Notify is called for an asynchronous signal, a Go signal handler will be installed for that signal. If, later, Reset is called for that signal, the original handling for that signal will be reinstalled, restoring the non-Go signal handler if any.

Go code built without -buildmode=c-archive or -buildmode=c-shared will install a signal handler for the asynchronous signals listed above, and save any existing signal handler. If a signal is delivered to a non-Go thread, it will act as described above, except that if there is an existing non-Go signal handler, that handler will be installed before raising the signal.

Windows

On Windows a ^C (Control-C) or ^BREAK (Control-Break) normally cause the program to exit. If Notify is called for os.Interrupt, ^C or ^BREAK will cause os.Interrupt to be sent on the channel, and the program will not exit. If Reset is called, or Stop is called on all channels passed to Notify, then the default behavior will be restored.

Additionally, if Notify is called, and Windows sends CTRL_CLOSE_EVENT, CTRL_LOGOFF_EVENT or CTRL_SHUTDOWN_EVENT to the process, Notify will return syscall.SIGTERM. Unlike Control-C and Control-Break, Notify does not change process behavior when either CTRL_CLOSE_EVENT, CTRL_LOGOFF_EVENT or CTRL_SHUTDOWN_EVENT is received - the process will still get terminated unless it exits. But receiving syscall.SIGTERM will give the process an opportunity to clean up before termination.

Plan 9

On Plan 9, signals have type syscall.Note, which is a string. Calling Notify with a syscall.Note will cause that value to be sent on the channel when that string is posted as a note.

func Ignore

```
func Ignore(sig ...os.Signal)
```

Ignore causes the provided signals to be ignored. If they are received by the program, nothing will happen. Ignore undoes the effect of any prior calls to Notify for the provided signals. If no signals are provided, all incoming signals will be ignored.

func Ignored

```
func Ignored(sig os.Signal) bool
```

Ignored reports whether sig is currently ignored.

func Notify

```
func Notify(c chan<- os.Signal, sig ...os.Signal)
```

Notify causes package signal to relay incoming signals to c. If no signals are provided, all incoming signals will be relayed to c. Otherwise, just the provided signals will.

Package signal will not block sending to c: the caller must ensure that c has sufficient buffer space to keep up with the expected signal rate. For a channel used for notification of just one signal value, a buffer of size 1 is sufficient.

It is allowed to call Notify multiple times with the same channel: each call expands the set of signals sent to that channel. The only way to remove signals from the set is to call Stop.

It is allowed to call Notify multiple times with different channels and the same signals: each channel receives copies of incoming signals independently.

func Reset

```
func Reset(sig ...os.Signal)
```

Reset undoes the effect of any prior calls to Notify for the provided signals. If no signals are provided, all signal handlers will be reset.

func Stop

```
func Stop(c chan<- os.Signal)
```

Stop causes package signal to stop relaying incoming signals to c. It undoes the effect of all prior calls to Notify using c. When Stop returns, it is guaranteed that c will receive no more signals.