Signals 1: Overview

1. **Overview**
   - **what are signals**
   - **signal terminology**
   - **disposition, handlers, generation**
   - **standard signals**
   - **key signals**
   - **signals and terminals**
   - **real-time signals**
2. System Calls
3. Signal Handlers
4. Synchronization
5. Signals and fork, exec, Pthreads
6. Advanced Topics
7. Signal Handler Issues

Signals

Signals are a mechanism for informing processes that some *event* has occurred.

They are effectively a type of **software interrupt**.

(An **interrupt** is an **asynchronous** notification that can literally interrupt the execution of a program or instruction.)

Most signals are sent to a process by the *kernel*.

However, it is also possible for a process to signal another process and even for a process to signal itself.

This allows signals to be used for process **synchronization** and (very simple) **IPC**.

Basic Signal Terminology

A signal is **generated** (or **raised**) by some event and will usually be destined for one process, though some events generate signals destined for multiple processes.

A generated signal is said to be **pending** before the kernel makes it available to a process.

When the kernel makes the signal available to a (running) process, it is said to be **delivered** and then **received** by the process.

How a process will respond to the delivery of a signal is called the signal's **disposition** (in the process).

A process may **block** particular signals, in which case they are held by the kernel (i.e., remain *pending*) until **unblocked**.

A process' **signal mask** indicates its currently *blocked* signals.

Signal Disposition

There are three possible *classes* of **signal dispositions**:

- **default** – take the **default action** for this signal
- **ignore** – *discard* the signal at delivery time (so no effect)
- **catch** – execute a function called a **signal handler**

There are five possible **default actions**:

- **ignore** – ignore the signal (discard it)
- **terminate** – terminate the process
- **core** – terminate the process and create a **core dump**
- **stop** – stop the process (job control)
- **continue** – continue the process (job control)

Signal Disposition (contd.)

When a signal is set to be *ignored*, the kernel actually just discards the signal at delivery time, so the process knows nothing about it and it has no effect on the process.

A **core dump** is a file containing the contents of the process address space, which can be used with `gdb` to examine the process state at termination.

Even if the *default disposition* is *ignore*, a different behavior may result from *explicitly* setting the disposition to ignore (e.g., for `SIGCHLD`).

Termination by a signal is considered **abnormal termination** (vs. **normal termination** as by calling `exit()`).

Signal Handlers

One of the unique aspects of signals is that signal *disposition* can be set so that a *user-defined function* gets run when a particular signal is delivered.

These functions are known as a **signal handlers**.

When a handler function gets run, signals are said to be **caught** or **handled**.

Signals can be delivered **asynchronously** (i.e., at any point in a program's execution).

Thus signal handlers provide *asynchronous execution* capability.

This can allow a program to respond immediately to some *event* (i.e., without having to **poll** to see if the event has occurred).

Signal Generation

Signals can be **generated** in several ways:

- hardware exceptions, such as:
  - divide by zero
  - illegal memory access
- OS software event, such as:
  - child process terminated (SIGCHLD)
  - writing to pipe with no open read ends (SIGPIPE)
- user types special characters at terminal, such as:
  - `ctrl-c` (*interrupt* character) (SIGINT)
- another process:
  - using `kill()` or `killpg()`
- the process itself:
  - using `abort()` or `alarm()`

Standard Signals

The term **standard signals** is commonly used to refer to **POSIX reliable signals**.

These are the most commonly used/seen signals.

Signals are identified by *positive integers*, but the portable method of denoting standard signals is with *symbolic names* like `SIGKILL`.

This is because the values for different standard signals may vary among UNIXes.

While you may have learned to to kill a process with the command "`kill -9` *PID*" using "`kill -SIGKILL` *PID*" is more portable (9 is the typical numeric value for SIGKILL).

See "`man 7 signal`" for info on the numeric values of signals.

Key Standard Signals

Here are the most important/common standard signals:

- **SIGTERM** – termination signal (`kill` default)
- **SIGKILL** – kill signal (cannot be *ignored* or *blocked*)
- **SIGINT** – "interrupt char" typed on terminal
- **SIGQUIT** – "quit char" typed on terminal
- **SIGCHLD** – child terminated (or stopped)
- **SIGSEGV** – invalid memory reference
- **SIGPIPE** – broken pipe (write to pipe with no readers)
- **SIGALRM** – timer signal (e.g., from `alarm()`)

Key Standard Signals (contd.)

continuing:

- **SIGHUP** – hangup detected on **controlling terminal** or termination of controlling process
- **SIGABRT** – abort signal from `abort()`
- **SIGUSR1** – user-defined signal 1
- **SIGUSR2** – user-defined signal 2
- **SIGILL** – illegal Instruction
- **SIGFPE** – floating point exception
- **SIGBUS** – bus error (bad memory access)
- **SIGPOLL** – pollable event (synonym for SIGIO)
- **SIGCONT** – continue if stopped
- **SIGSTOP** – stop process

Key Standard Signals (contd.)

The *default disposition* for most of these symbols is *termination* or *termination with core dump*.

The following signals have *ignore* as their *default dispositions*: `SIGCHLD` and `SIGURG`.

The following signals have *continue/stop* as their *defaults*: `SIGCONT` and `SIGSTOP` (relates to **job control**).

Signals that result from CPU/hardware issues are referred to as **synchronous signals**: SIGBUS, SIGFPE, SIGILL, SIGSEGV.

Synchronous signals are **thread-specific** (see more later).

Properties of Standard Signals

All standard signals—except for two—can be *ignored or blocked* by processes.

The two signals that *cannot be ignored or blocked* are: `SIGKILL` and `SIGSTOP`.

This is why one is told to do "`kill -SIGKILL` *PID*" instead of just "`kill` *PID*": the default `SIGTERM` signal can be blocked/ignored in processes so process may not terminate.

Using `-SIGKILL` (or `-9`) ensures process will be terminated.

(Note that *stopped processes* cannot be terminated by `SIGTERM`.)

Properties of Standard Signals (contd.)

When a signal is *generated* for a process that has that signal *blocked*, the signal is **queued** by the kernel for possible later delivery (if eventually *unblocked*).

Queueing of standard signals has two key limitations:

- at most *one instance* of each standard signal can be queued
- the *order* queued signals are *delivered* once unblocked is *indeterminate*

This means that if a process blocks signals for a period of time and then unblocks them, the process cannot determine how many of each signal was generated during the block period, nor what order signals were generated in during that period.

(**Real-time signals** address these limitations—see below.)

Signals and Terminals

By default, the Linux/UNIX **terminal driver** recognizes a number of *special characters* to interrupt running programs, do command line editing, send EOF, etc.

Instead of passing special characters through to programs, the terminal driver takes other actions.

Three of the terminal driver special characters cause the terminal driver to *generate signals* and send them to processes associated with the terminal.

This is actually one of the most common ways that users interact with signals (though they are often not aware of this).

Signals and Terminals (contd.)

The terminal driver special characters that generate signals are:

- **INTR** – set to `ctrl-c` by default, when the user types this character, the terminal driver sends a `SIGINT` signal
- **QUIT** – set to `ctrl-\` by default, when the user types this character, the terminal driver sends a `SIGQUIT` signal
- **SUSP** – set to `ctrl-z` by default, when the user types this character, the terminal driver sends a `SIGTSTP` signal

In all cases, the signals are sent to all processes in the **foreground process group** (for which the terminal is a **controlling terminal**).

(Special characters and other terminal characteristics can be changed with the `stty` command or `tcsetattr()` syscall.)

Real-Time Signals

So far we have discussed **standard signals**—i.e., **POSIX reliable signals**.

These are the most commonly seen/used types of signals.

Linux also supports the newer **POSIX real-time signals**.

Real-time signals address some key limitations of standard signals.

Unlike standard signals, real-time signals have no predefined uses (and no symbolic names).

This means that the entire set of real-time signals can be used for application-specific purposes.

The **default disposition** for real-time signals is to *terminate* the receiving process.

Real-Time Signals (contd.)

As with standard signals, real-time signals are represented by positive integers.

The range of supported real-time signals is defined by the macros `SIGRTMIN` and `SIGRTMAX`.

Linux supports 33 real-time signals (32 to 64), the glibc Pthreads implementation uses two or three of these internally, so adjusts `SIGRTMIN` suitably (to 34 or 35).

Because the range of available real-time signals can vary among Linux/UNIX systems, programs should *never refer to real-time signals using hard-coded numbers*.

Instead, programs should always refer to real-time signals using the notation `SIGRTMIN+n` (and possibly include run-time checks that `SIGRTMIN+n` does not exceed `SIGRTMAX`).

Real-Time Signals (contd.)

Real-time signals address two limitations of standard signals:

- they are *fully queued:* multiple instances of each real-time signal can be queued while blocked
- they are delivered in a *guaranteed order:* queued real-time signals of the same type are delivered in the order they were generated, different queued real-time signals are delivered starting with the *lowest numbered* (i.e., lower numbered real-time signals have higher "priority")

Note that if both standard and real-time signals are pending for a process, Linux gives priority to standard signals (though POSIX does not require this).