Architecture et Systèmes

Stefan Schwoon

Cours L3, 2014/15, ENS Cachan

Signals

Signals define a (very basic) interface provided by the system through which processes can interact.

A signal is sent to some process *A* by either the kernel or some other process *B*.

$\Rightarrow$ typically used to handle low-level tasks;

$\Rightarrow$ not meant for complex communication/exchanging data.

Meant to indicate specific conditions (error, interruption, alarm, . . .).

Terminology

A signal is a system-defined message.

List of defined signals: `kill -l` on the command line.

E.g., Linux supports 64 signals, of which around 30 have a defined meaning.

A signal transfer happens in two steps:

Sending: The signal is sent to process *A* (it becomes *pending*).

Delivery: the system makes process *A* react to a pending signal.

Types of signals

Some examples:

SIGINT – generated by typing Ctrl+C in the console.

SIGTERM – sent to each process when the system is about to shut down.

SIGKILL – kills process definitely, cannot be overridden.

SIGUSR1 – user-defined signal

SIGTSTP – generated by typing Ctrl+Z in the console.

SIGALRM – used by sleep(3), alarm(2)

Sending

Suppose process *B* wants to send signal *s* to process *A*.

B calls the kill(2) function. E.g., *kill(1000,9)* sends signal 9 to process id 1000.

The system memorises that *A* has received a signal of type *s*.

Some restrictions:

> *A* and *B* must be owned by the same user, or *A* must belong to the administrator.
>
> A process can have only one signal of a type pending at the same time. Subsequent signals of the same type are discarded.

See also: kill(1) (command line program)

Delivery

When *A* is next scheduled for execution, the system first checks whether there is a pending system.

If so, then the signal is first *delivered*. Normal process execution continues after the delivery.

For each signal type, *A* has a current disposition determining what should happen upon delivery.

Possible dispositions: Ign (ignore signal), Term (terminate process), Core (terminate and memory dump), Stop/Cont (stop/continue process), or a user-defined signal handler.

See also `signal(7)`.

Signal handlers

A process can change its disposition for a signal with sigaction(2) or signal(2).

Some dispositions cannot be changed (for instance, for SIGKILL).

A user-defined *signal handler* is a pointer to a function to be executed upon delivery of that signal.

When a processes forks, the child inherits the dispositions from its parent. However, `exec` will reset dispositions for all user-defined handlers to their defaults.

Blocking signals

Each process also possesses a blocking mask
$\Rightarrow$ meant to *temporarily* block delivery of signals.

Signals can be added and removed from the mask using sigprocmask(2).

Any signal contained in the blocking mask will remain pending until it is removed from the mask.

Signals can also be automatically blocked during execution of a signal handler, this can be specified in sigaction.

sigsuspend(2) allows to temporarily replace the signal mask until a signal is received, then immediately restore the mask.

Caveats

Attention: Certain system calls can be interrupted by signals!

Examples:

`wait` returns when a child has terminated or a signal delivery intervenes.

`read` (and derived functions like `getchar`, ...) also exceptionally terminate when a signal intervenes.

So in principle, it is always necessary to check the return code of these functions to be sure...

Process groups

Each process belongs to a process group.

Process groups have a numerical identifier
(usually identical to the PID of some group member).

The `setpgid` call changes the group of a given process. Examples:

`setpgid(p,g)` – makes *g* the new group of process *p*

`setpgid(0,0)` – equivalent to `setpgid(p,p)`, where *p* is the PID of the caller

Obvious restrictions apply concerning the identity of the calling process, *p*, and *g*...

Signals and process groups

Signals can be sent to a single process (as discussed) or to all processes inside a group.

When `kill` receives a negative argument for PID, it interprets it as a group ID.

Example: `kill(SIGINT,-100)` – send SIGINT to process group 100

Works with both the command-line version and the system call `kill`.

Process groups in the shell

A shell is running inside a terminal.
Historically: an actual hardware device with screen/keyboard;
nowadays: a console window or a screen running in text mode (try Alt-Ctrl-F1 on a Linux machine).

The terminal has a notion of *foreground process group*.

Keyboard input (and signals like from Ctrl-C) are sent to said foreground process group.

When launching a command, the shell forks, then makes it child a new foreground process group, then execs the command.

Foreground/background jobs

Typing Ctrl+Z in the terminal will send SIGTSTP to the foreground processes.

Default behaviour: Processes stop, shell takes over.

Typing `bg` or `fg` sends SIGCONT to these processes, allowing them to continue as either background or foreground processes.

In fact, the shell can handle multiple background groups if Ctrl+Z is used several times...

Note: Ctrl+S and Ctrl+Q suspend/resume output in the terminal, without actually stopping the processes.