CS415 Module 2 Part D - Signals

Athens State University

Contents

1 Introduction to Signals

What are signals?

- **Signal**: A notification to a process that an event has occurred
  - Sometimes known as software interrupts because they interrupt the normal execution flow of a program
  - One cannot, in most cases, predict when a signal will arrive
- A process can, given appropriate permissions, send a signal to another process

What can cause a signal?

- A hardware exception occurred and the kernel sent a corresponding signal to processes needing to know that fact
- The user enters a special keyboard combination such as the *interrupt character* (usually the ASCII Control-C character) or the *suspend character* (usually the ASCII Control-Z character)
- Some software event has occurred such as input available on a file descriptor, a terminal window size changed, or a child process terminated

Each signal is defined as a unique (and hopefully small) integer starting from 1. These integers are defined in the Linux header file `signal.h` and you should use the symbolic names found in that file as signal numbers are permitted to vary across implementations.

Signal names in `signal.h` take the form `SIGxxxx`. For example, typing the interrupt character results in the kernel sending a `SIGINT` (signal 2) to the current process.

Traditional vs. Realtime Signals

- The traditional (or standard) signals are defined in the POSIX standard to be signals 1 through 31
- Realtime signals are an extension to the traditional standard signals for application use
  - The standard POSIX signals only defined two signals for application use: `SIGUSR1` and `SIGUSR2`
  - These signals, unlike standard signals, are queued and can pass references to additional data
  - Our discussion will focus solely on traditional signals

Terms to remember

- Signals are *generated* by some event
- Generated signals are *delivered* to a process
- *Pending* signals are signals that have been generated but have not yet been delivered
  - Pending signals are delivered to a process as soon as it next scheduled to run
  - We can *block* a signal from being delivered to a process by adding that signal to that process's *signal mask*

The signal mask is a bitmap stored in the process control block. Signals blocked by a process remain pending until the target process removes the signal from its signal mask. System calls are provided to allow for adding or removing a signal from the signal mask.

What happens when a signal is delivered?

- There is a set of default actions that can happen:
  - The signal can be *ignored*.
  - The process is `terminated`.
  - A `core dump file` is generated and the process is terminated
  - The process is forcibly suspened
  - The process `resumes` execution after being previously stopped

If a signal is ignored, then the signal is discarded by the kernel and has not effect on the process.

A synonym for terminated is *abnormal process terminated* or `abnormal process end`. This term, esp. for old-timers such as your Glorious Instructor (think Dr. Lewis doing his best (or worst, depending) Dr. Evil imitation) know as an *abend* in your process.

Core dump files are extremely useful as they contain a snapshot of the virtual memory of the process at the time of the signal was delivered. Debuggers such as GDB, LLDB, and DDX use this information to permit users to examine the state of the process at the time it terminated.

Signal Disposition

- **Disposition** Overriding the the default action for a signal
- Types of dispositions
  - Resetting the disposition to the *default action*
  - Set the signal to be ignored
  - Execute a program provided *signal handler*

Signal handlers are user-defined functions that perform application tasks in response to the delivery of a signal. For example, the Bash shell has a handler for the `SIGINT` signal that causes it to terminate a running process and return control to shell's main input loop.

We say that we are *installing* or *establishing* a signal handler when we call a special system call that remaps the signal to call the signal handler function. When a signal handler function is involved by delivery of a signal, then the signal has been *handled* or *caught*.

Note that we are unable to set a process to terminate or dump core in response to signal. We must set a signal handler to catch the signal of interest and have that signal handler explicitly call either the `exit()` or `abort()` system calls. The `abort()` system call generates a `SIGABRT` signal where the default action for that signal is to force the process to dump core and terminate.

A side note about Linux: information about a process is stored within the `/proc` file system. One can example the `/proc/PID/status` file to determine how a process is currently configured to respond to signals. Bitmasks are displayed in hexadecimal with least significant bit representing signal 1.

The Linux Standard Signals

Table 20-1: Linux signals

Name	Signal number	Description	SUSv3	Default
SIGABRT	6	Abort process	•	core
SIGALRM	14	Real-time timer expired	•	term
SIGBUS	7 (SAMP=10)	Memory access error	•	core
SIGCHLD	17 (SA=20, MP=18)	Child terminated or stopped	•	ignore
SIGCONT	18 (SA=19, M=25, P=26)	Continue if stopped	•	cont
SIGEMT	undef (SAMP=7)	Hardware fault		term
SIGFPE	8	Arithmetic exception	•	core
SIGHUP	1	Hangup	•	term
SIGILL	4	Illegal instruction	•	core
SIGINT	2	Terminal interrupt	•	term
SIGIO / SIGPOLL	29 (SA=23, MP=22)	I/O possible	•	term
SIGKILL	9	Sure kill	•	term
SIGPIPE	13	Broken pipe	•	term
SIGPROF	27 (M=29, P=21)	Profiling timer expired	•	term
SIGPWR	30 (SA=29, MP=19)	Power about to fail		term
SIGQUIT	3	Terminal quit	•	core
SIGSEGV	11	Invalid memory reference	•	core
SIGSTKFLT	16 (SAM=undef, P=36)	Stack fault on coprocessor		term
SIGSTOP	19 (SA=17, M=23, P=24)	Sure stop	•	stop
SIGSYS	31 (SAMP=12)	Invalid system call	•	core
SIGTERM	15	Terminate process	•	term
SIGTRAP	5	Trace/breakpoint trap	•	core
SIGTSTP	20 (SA=18, M=24, P=25)	Terminal stop	•	stop
SIGTTIN	21 (M=26, P=27)	Terminal read from BG	•	stop
SIGTTOU	22 (M=27, P=28)	Terminal write from BG	•	stop
SIGURG	23 (SA=16, M=21, P=29)	Urgent data on socket	•	ignore
SIGUSR1	10 (SA=30, MP=16)	User-defined signal 1	•	term
SIGUSR2	12 (SA=31, MP=17)	User-defined signal 2	•	term
SIGVTALRM	26 (M=28, P=20)	Virtual timer expired	•	term
SIGWINCH	28 (M=20, P=23)	Terminal window size change		ignore
SIGXCPU	24 (M=30, P=33)	CPU time limit exceeded	•	core
SIGXFSZ	25 (M=31, P=34)	File size limit exceeded	•	core

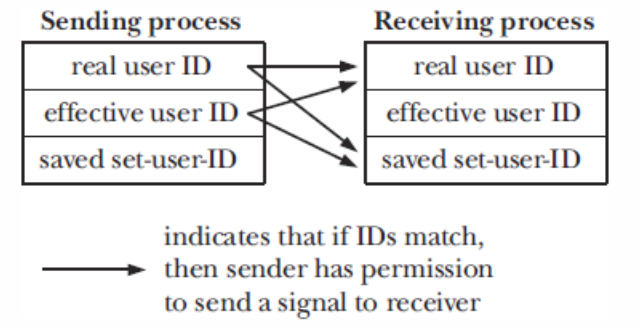
Abbreviations in the signal number column: Solaris (S), DEC Alpha (A), MIPS (M) HP PA-RISC (P) (that should tell you the age of this chart).

Sending signals: the `kill` command and system call

```
1 #include signal.h
  int kill(pid_t pid, int sig)
```

- We use `kill` as this default signal action in Classical UNIX was to terminate the process
- Return a value of 0 on success or -1 on failure
- The value of `pid` is interpreted by `kill()` in the same fashion as the `pid` parameter in `waitforpid()`

Required permissions to use `kill()`



- A privileged process may send a signal to any process
- The `init` process is a special case
- Other processes can only send signals to other processes with matching user-ids

The `init` process runs as the `root` user, the system superuser or administrator. Given the rather bad that could happen should the root user attempt to terminate this process, `init` is configured to only respond to signals for which the code has explicit signal handlers.

Checking for the existence of a process

- Passing a signal value of 0 to the *kill()* system call tells the kernel to only check to see if the process can be signaled
  - We can use a positive response from this call to indicate the process exists
  - Note this doesn't mean the process is running or even running the same program as before
- Other ways to check for existence of a process
  - Use the `wait()` system calls
  - Use the semaphore and exclusive file lock IPC system calls (more later)
  - Check to see if there is an entry for a PID in the `/proc` file system

2 Key Points

Key Points

- What are signals?
- How are they processed?
- What can we do with them?