Signals: Asynchronous Events

August 25, 2017

Signals: asynchronous events

Linux/Unix signals are a type of event. Signals are asynchronous in nature and are used to inform processes of certain events happening.
Examples:

- User pressing the interrupt key (usually `Ctl-c` or `Delete` key). Generates the `SIGINT` signal.
- User pressing the stop key (usually `Ctl-z`). Generates the `SIGTSTP` signal, which stops (suspends) the process.
- The signal `SIGCONT` can restart a process if it is stopped.
- Signals are available for alarm (`SIGALRM`), for hardware exceptions, for when child processes terminate or stop and many other events.
- Special signals for killing (`SIGKILL`) or stopping (`SIGSTOP`) a process. These cannot be ignored by a process.

POSIX signals list

Read `man signal` and `man 7 signal` for more information.

```
SIGHUP   Hangup detected on controlling terminal
         or death of controlling process
SIGINT   Interrupt from keyboard
SIGQUIT  Quit from keyboard
SIGILL   Illegal Instruction
SIGABRT  Abort signal from abort
SIGFPE   Floating point exception
SIGKILL  Kill signal
SIGSEGV  Invalid memory reference
SIGPIPE  Broken pipe: write to pipe with no readers
SIGALRM  Timer signal from alarm
SIGTERM  Termination signal
SIGUSR1  User-defined signal 1
SIGUSR2  User-defined signal 2
SIGCHLD  Child stopped or terminated
SIGCONT  Continue if stopped
SIGSTOP  Stop process
SIGTSTP  Stop signal from keyboard
SIGTTIN  tty input for background process
SIGTTOU  tty output for background process
```

Signals (contd.)

- For each signal there are three possible actions: default, ignore, or catch. The system call `signal()` attempts to set what happens when a signal is received. The prototype for the system call is:

```
void (*signal(int signum, void (*handler)(int)))(int);
```

- The above prototype can be made easier to read with a typedef as shown below.

```
typedef void sighandler_t(int);
sighandler_t *signal(int, sighandler_t *);
```

- The header file `<signal.h>` defines two special dummy functions `SIG_DFL` and `SIG_IGN` for use as signal catching actions. For example:

```
signal(SIGALRM, SIG_IGN);
```

To kill or to really kill?

- The system call `kill()` is used to send a specified signal to a specified process. For example:

```
kill(getpid(), SIGKILL);
kill(getpid(), SIGSTOP);
kill(pid, SIGCONT);
```

- Special signals for killing (`SIGKILL`) or stopping (`SIGSTOP`) a process. These cannot be ignored by a process.
- Linux has a command named `kill` that invokes the `kill()` system call.

```
kill -s signal pid
kill -l  --> list all signals
kill -9  --> send SIGKILL
```

- To kill a process use `kill -1` (`SIGHUP`) or `kill -15` (`SIGTERM`) first to give the process a chance to clean up before being killed (as those signals can be caught). If that doesn't work, then use `kill -9` to send `SIGKILL` signal that cannot be caught or ignored. In some circumstances, however, even `SIGKILL` doesn't work....

kill -9

Because I could not stop for Death,
He kindly stopped for me;
The carriage held but just ourselves
And Immortality.
...
Emily Dickinson

Examples

- A simple signal handler example. Uses the `alarm` system call and signal handler. signals/wake-up.c
- A signal handler example that ignores Ctrl-c and Ctrl-z (and prints an annoying message). signals/sig-blocker.c
- A signal handler example that catches a segmentation fault! signals/catch-null-ptr.c
- A bigger example of systems programming. Sets a time limit on a process. signals/timeout.c

In-Class exercise

- **Autosave?** Consider the following C program sketch. Choose the right answer that explains what the code does.

```
int main() {
    /* ... */
    signal(SIGALRM, savestate);
    alarm(10);
    /* ... */
    for (;;) {
        /* do something */
    }
}
void savestate(int signo) {
    /* save the state to disk */
}
```

1. Saves the state of the program to disk every 10 seconds
2. Exits after saving the state of the program once to the disk after 10 seconds
3. Keeps running after saving the state of the program once to the disk after 10 seconds
4. Exits after saving the state of the program twice to the disk after 10 seconds
5. Never saves the state of the program to the disk