Advanced Programming in the UNIX Environment

Week 07, Segment 4:
Signals

Department of Computer Science
Stevens Institute of Technology

Jan Schaumann
jschauma@stevens.edu
https://stevens.netmeister.org/631/

Signals summarized

Signals

Signals are a way for a process to be notified of *asynchronous* events. Some examples:

- the user hits Ctrl+C (`SIGINT`)
- the user hits Ctrl+Z (`SIGTSTP`)
- a background process attempts I/O on the controlling terminal (`SIGTTOU` / `SIGTTIN`)
- a timer you set has gone off (`SIGALRM`)
- a user disconnected from the system (`SIGHUP`)
- a user resized the terminal window (`SIGWINCH`)
- ...

See also: `signal(2)`/`signal(3)`/`signal(7)` (Note: these man pages vary across platforms!)

Signals

Other ways signals might be generated:

- terminal generated signals (user presses a key combination which causes the terminal driver to generate a signal)
- hardware exceptions (divide by 0, invalid memory references, etc.)
- software conditions (other side of a pipe no longer exists, urgent data has arrived on a network file descriptor, etc.)
- `kill(1)` allows a user to send any signal to any process (if the user is the owner or superuser)
- `kill(2)`, the system call

kill(2)

```
#include <signal.h>

int kill(pid_t pid, int sig);
```

Returns: 0 on success, -1 otherwise

- if `pid` > 0, then the signal is sent to the process whose PID is `pid`
- if `pid` == 0, then the signal is sent to all processes whose process group ID equals the process group ID of the sender
- if `pid` == `-1`, then (not POSIX, but BSD and Linux):
    - if `euid` == 0, the signal is sent to all processes excluding system processes and the process sending the signal
    - else, the signals is sent to all processes with the same uid as the user excluding the process sending the signal
- if `pid` < `-1`, then (Linux) the signal is sent to every process in the process group whose ID is -`pid`
- if `sig` == 0, no signal is sent, but error checking is performed (*i.e.*, an easy way to test "does `pid` exist")

Signals

Once we receive a signal, we can do one of several things:

- *Accept the default*. Have the kernel do whatever is defined as the default action for this signal.
- *Ignore it.* (Note: there are some signals which we CANNOT or SHOULD NOT ignore.)
- *Catch it.* That is, have the kernel call a function which we define whenever the signal occurs.
- *Block it.* The delivery of the signal is postponed until it is unblocked.

signal(3)

```
#include <signal.h>
void (*signal(int sig, void (*func)(int)))(int);
```

Returns: previous signal handler if ok, SIG_ERR otherwise

func can be:

- SIG_IGN, which requests that we ignore the signal signo
- SIG_DFL, which requests that we accept the default action for signal signo
- a pointer to a function to invoke when the signal is received

On some systems/versions, after a signal handler was executed, it was *reset* to SIG_DFL.

signal(3)

```
#include <signal.h>
typedef void (*sig_t)(int);
sig_t signal(int sig, sig_t func);
```

Returns: previous signal handler if ok, SIG_ERR otherwise

func can be:

- SIG_IGN, which requests that we ignore the signal signo
- SIG_DFL, which requests that we accept the default action for signal signo
- a pointer to a function to invoke when the signal is received

On some systems/versions, after a signal handler was executed, it was *reset* to SIG_DFL.

signal(3)

```
#include <signal.h>
void (*signal(int sig, void (*func)(int)))(int);
```

Returns: previous signal handler if ok, SIG_ERR otherwise

func can be:

- SIG_IGN, which requests that we ignore the signal signo
- SIG_DFL, which requests that we accept the default action for signal signo
- a pointer to a function to invoke when the signal is received

On some systems/versions, after a signal handler was executed, it was *reset* to SIG_DFL.

Terminal — 130×24

```
        if (signal(SIGHUP, sig_usr) == SIG_ERR) {
                err(EXIT_FAILURE, "unable to catch SIGHUP");
                /* NOTREACHED */
        }

        (void)printf("%d\n", getpid());
        for ( ; ; ) {
                pause();
                /* Note that the compiler is smart enough to rea
lize we
                 * don't return from main, so won't warn that we
 didn't
                 * return a value. */
        }
}
apue$ cc -Wall -Werror -Wextra sigusr.c
apue$ ./a.out
1021
Nobody expects SIGUSR1!
A surprising turn of events occurred!
received signal: 1
apue$
```

0 sh

```
apue$ kill -USR1 1021
apue$ kill -USR2 1021
apue$ kill -INFO 1021
apue$ kill -HUP 1021
apue$
```

1 sh

sigaction(2)

```
#include <signal.h>
int sigaction(int sig, const struct sigaction * restrict act, struct sigaction * restrict fact);
```

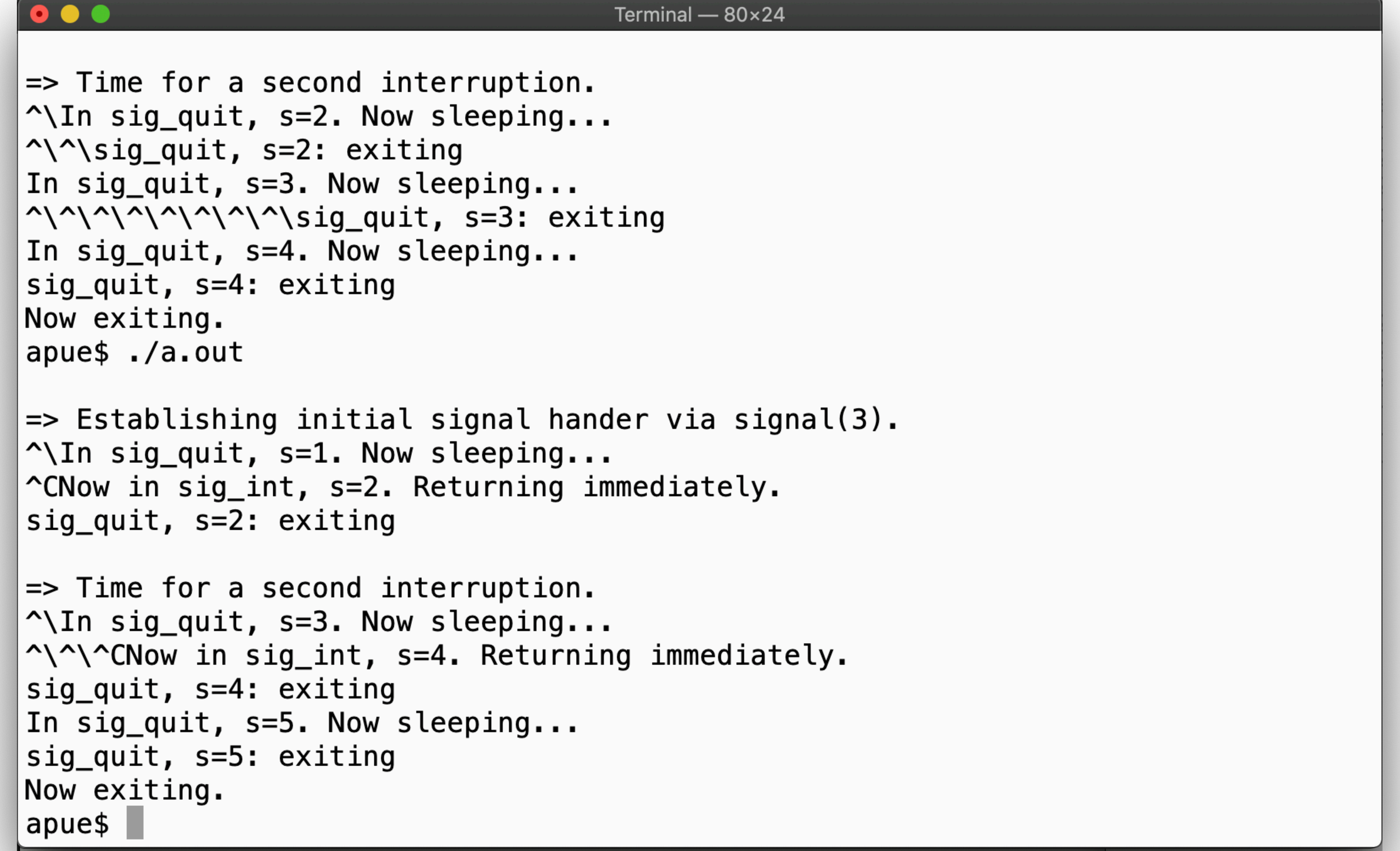
Returns: 0 if ok, -1 otherwise

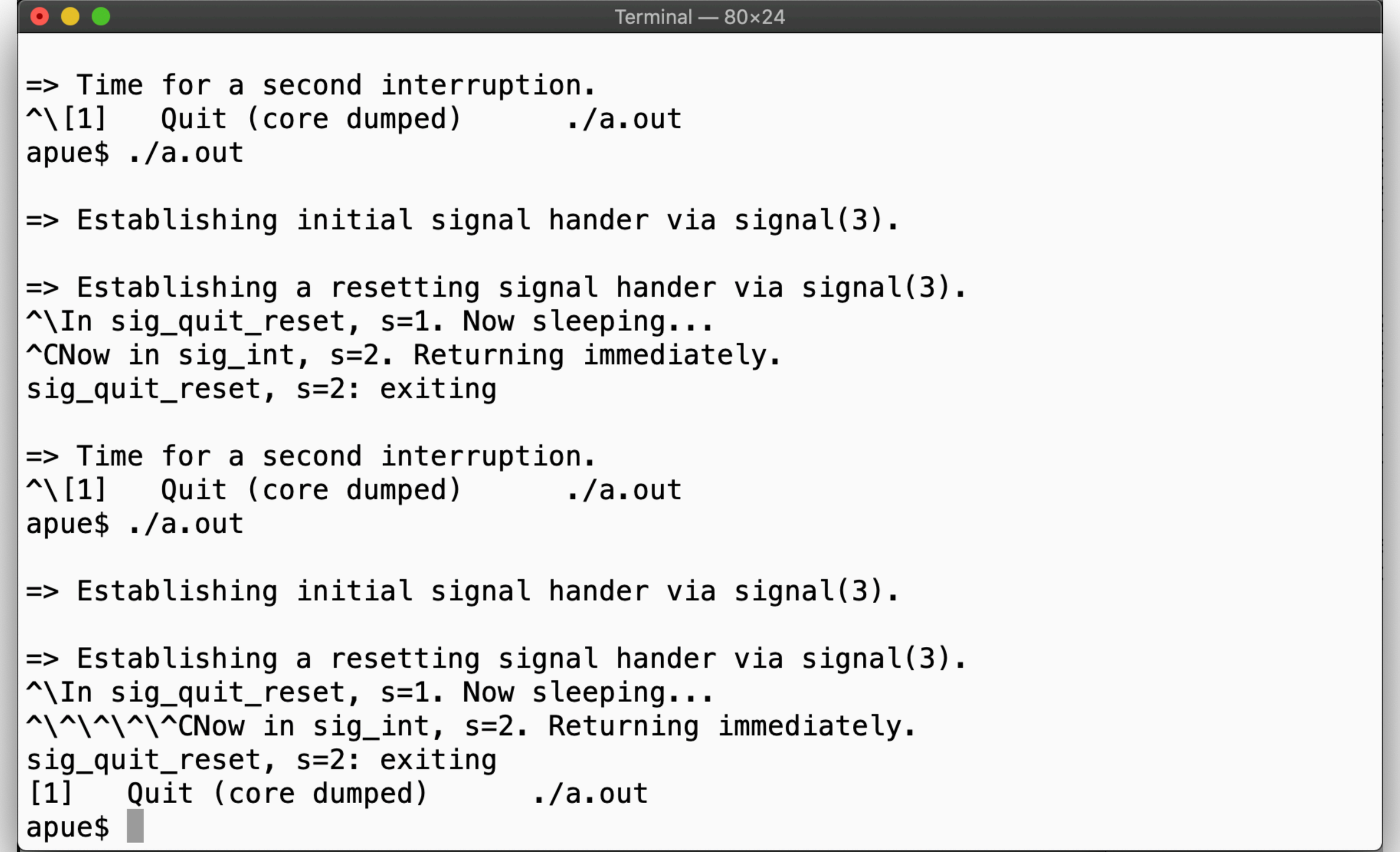
signal(3) is (nowadays) commonly implemented via sigaction(2):

```
struct sigaction {
        void        (*_sa_handler)(int sig);
        void        (*_sa_sigaction)(int sig, siginfo_t *info, void *ctx);
        sigset_t    sa_mask;
        int         sa_flags;
}
```

More detailed signal concepts

- While executing a signal handler, the signal that triggered it is blocked, but other signals may occur!
- Blocked signals are marked as *pending*; you can inspect the set of pending signals.
- The signal that triggered you entering a signal handler is automatically *unblocked* upon exit from that signal handler, meaning it will be delivered and you will re-enter the same handler right away.
- Multiple signals of the same type arriving in sequence while being blocked may be merged.
- After a `fork(2)`, all signal dispositions and signal masks remain the same in the child as in the parent.
- The `execve(2)` system call reinstates the default action for all signals which were caught, but ignored signals continue to be ignored; the signal mask also remains the same.


Terminal — 80×24
=> Time for a second interruption.
^\In sig_quit, s=2. Now sleeping...
^\^\sig_quit, s=2: exiting
In sig_quit, s=3. Now sleeping...
^\^\^\^\^\^\^\^\^\sig_quit, s=3: exiting
In sig_quit, s=4. Now sleeping...
sig_quit, s=4: exiting
Now exiting.
apue$ ./a.out
=> Establishing initial signal hander via signal(3).
^\In sig_quit, s=1. Now sleeping...
^CNow in sig_int, s=2. Returning immediately.
sig_quit, s=2: exiting
=> Time for a second interruption.
^\In sig_quit, s=3. Now sleeping...
^\^\^CNow in sig_int, s=4. Returning immediately.
sig_quit, s=4: exiting
In sig_quit, s=5. Now sleeping...
sig_quit, s=5: exiting
Now exiting.
apue$


Terminal — 80×24
=> Time for a second interruption.
^\[1]   Quit (core dumped)   ./a.out
apue$ ./a.out
=> Establishing initial signal hander via signal(3).
=> Establishing a resetting signal hander via signal(3).
^\In sig_quit_reset, s=1. Now sleeping...
^CNow in sig_int, s=2. Returning immediately.
sig_quit_reset, s=2: exiting
=> Time for a second interruption.
^\[1]   Quit (core dumped)   ./a.out
apue$ ./a.out
=> Establishing initial signal hander via signal(3).
=> Establishing a resetting signal hander via signal(3).
^\In sig_quit_reset, s=1. Now sleeping...
^\^\^\^\^CNow in sig_int, s=2. Returning immediately.
sig_quit_reset, s=2: exiting
[1]   Quit (core dumped)   ./a.out
apue$

```
Terminal — 80×24

=> Unblocking SIGQUIT...
In sig_quit, s=1. Now sleeping...
^\sig_quit, s=1: exiting
In sig_quit, s=2. Now sleeping...
sig_quit, s=2: exiting
SIGQUIT unblocked - sleeping some more...
^\In sig_quit, s=3. Now sleeping...
sig_quit, s=3: exiting
Now exiting.
apue$ ./a.out 1

=> Establishing initial signal hander via signal(3).

=> Blocking delivery of SIGQUIT...

=> Now going to sleep for 5 seconds...
^\^\^\^\^\^\^\
=> Checking if any signals are pending...

=> Unblocking SIGQUIT...
SIGQUIT unblocked - sleeping some more...
Now exiting.
apue$
```

Signals

- Signals are notifications of asynchronous events; delivery is similar to a hardware interrupt (current context is saved, a new context is built).
- Signals may be allowed to take the default action (`SIG_DFL`), be ignored (`SIG_IGN`), caught (`signal(3)` / `sigaction(2)`), or blocked (`sigprocmask(2)`).
- Arriving signals may be merged, then immediately delivered; different signals may interrupt the current signal handler.
- A number of additional options are available; see `sigaction(2)`.

- If your signal handler can be interrupted, what library functions or system calls are safe to use within a signal handler?