

# CS810 - Advanced Programming in the UNIX Environment

—

## Signals

---

Department of Computer Science  
Stevens Institute of Technology  
Jan Schaumann

`jschauma@cs.stevens.edu`

`http://www.cs.stevens.edu/~jschauma/810-APUE/`

## Signal Concepts

---

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

- a timer you set has gone off (SIGALRM)
- some I/O you requested has occurred (SIGIO)
- a user resized the terminal "window" (SIGWINCH)
- a user disconnected from the system (SIGHUP)

As you can see, all signals are given descriptive abbreviations which begin with SIG. Note that before making use of these signals, you should check the system manual pages on your particular system for exact descriptions.

## Signal Concepts

---

These signal names are defined in `<signal.h>`. Besides the asynchronous events listed previously, there are many ways to generate a signal:

- 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)
- `kill(1)` allows a user to send any signal to any process (if the user is the owner or superuser)
- `kill(2)` (a function call, not the unix command) performs the same task
- software conditions (other side of a pipe no longer exists, urgent data has arrived on a network file descriptor, etc.)

## Signal Concepts

---

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

- 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.
- Accept the default. Have the kernel do whatever is defined as the default action for this signal

## signal(3)

---

```
#include <signal.h>
```

```
void (*signal(int signo, void (*func)(int)))(int);
```

Returns: previous disposition of signal if OK, SIG\_ERR otherwise

## signal(3)

---

```
#include <signal.h>

void (*signal(int signo, void (*func)(int)))(int);

Returns: previous disposition of signal if OK, SIG_ERR otherwise
```

To simplify that prototype, we can use:

```
typedef void Sigfunc(int);
```

So that the declaration becomes:

```
Sigfunc *signal(int signo, Sigfunc * func);
```

Now the func argument 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*
- or the address of a function which should catch or handle a signal

## Program Startup

---

When a program is `execed`, the status of all signals is either *default* or *ignore*.

## Program Startup

---

When a program is `execed`, the status of all signals is either *default* or *ignore*.

When a process `fork(2)s`, the child inherits the parent's signal dispositions.

## Program Startup

---

When a program is `execed`, the status of all signals is either *default* or *ignore*.

When a process `fork(2)s`, the child inherits the parent's signal dispositions.

A limitation of the `signal(3)` function is that we can only determine the current disposition of a signal by *changing* the disposition.

## Interrupted System Calls

---

Some system calls can block for long periods of time (or forever). These include things like:

- `read(2)`s from files that can block (pipes, networks, terminals)
- `write(2)` to the same sort of files
- `open(2)` of a device that waits until a condition occurs (for example, a modem)
- `pause(3)`, which purposefully puts a process to sleep until a signal occurs

Traditionally, if a process was executing one of these calls and a signal occurred, the call was aborted with an `errno` return of `EINTR`. The assumption being that if something generated a signal for this process, it's important enough to stop waiting for whatever it is that the process was doing.

This behavior necessitated checking failed system calls of this nature explicitly for this condition, and restarting the system call. Because of this, many implementations of Unix automatically restart certain system calls.

## Reentrant Functions

---

If your process is currently handling a signal, what functions are you allowed to use? Consider that whenever a signal occurs, control is immediately transferred from whatever current instruction is being executed to the start of your signal handler.

Suppose your program was in the middle of a malloc system call when a signal occurs. Can you call malloc in your signal handler? Most certainly not. The state of the data structures used by malloc to track allocated memory are not in a consistent state if you interrupt its operation. Calling malloc again before you allow the first instance of malloc to complete would be disastrous.

Functions which can have several instances in progress simultaneously are said to be reentrant functions. POSIX.1 guarantees that certain functions are reentrant. A list of them can be found on p. 306 of your text.

## kill(2) and raise(3)

---

```
#include <sys/types.h>
#include <signal.h>

int kill(pid_t pid, int signo);
int raise(int signo);
```

- $pid > 0$  – signal is sent to the process whose PID is  $pid$
- $pid == 0$  – signal is sent to all processes whose process group ID equals the process group ID of the sender
- $pid == -1$  – POSIX.1 leaves this undefined, BSD defines it (see `kill(2)`)

## sigaction(2)

---

```
#include <signal.h>

int sigaction(int signo, const struct sigaction *act, struct sigaction *oact);
```

This function allows us to examine or modify the action associated with a particular signal.

```
struct sigaction {
    void (*sa_handler)();    /* addr of signal handler, or
                             SIG_IGN or SIG_DFL */
    sigset_t sa_mask;       /* additional signals to block */
    int sa_flags;           /* signal options */
};
```