

# CSCI 4061: Signals and Signal Handlers

Chris Kauffman

*Last Updated:  
Mon Mar 8 03:54:07 PM CST 2021*

# Signals Overview

- ▶ Signals are an old system of communication to convey a limited amount of info to a process
- ▶ “Delivered” by the OS to a running process to inform of it of an event
- ▶ Process responds in one of several ways according to its **Disposition**
- ▶ Default disposition for most signals is to die
- ▶ Process can **ignore** most signals or handle them with a designated functions but some cannot be **caught** in this way
- ▶ Signals are **asynchronous**, could delivered to a process **at any time** which makes them a pain in the @\$\$

# Sending Signals: kill Utility and kill() Syscall

Have seen that the kill utility sends signals on the command line

```
> kill 1234          # attempt to end a program "nicely"
> kill -TERM 1234    # same as above
> kill -15 1234      # same as above
> kill -9 1234       # forcefully kill a program
> kill -KILL 1234    # same as above
```

Corresponds to invocations of the kill() system call

```
> man 2 kill
```

NAME

kill - send signal to a process

SYNOPSIS

```
#include <sys/types.h>
#include <signal.h>
int kill(pid_t pid, int sig);
```

DESCRIPTION

The kill() system call can be used to send any signal to any process group or process.

...

EXAMPLE

```
{ ...
    kill(1234, SIGKILL);
}
```

# Process Signal Disposition

```
> man 7 signal
```

```
...
```

## Signal dispositions

Each signal has a current disposition, which determines how the process behaves when it is delivered the signal.

The entries in the "Action" column of the tables below specify the default disposition for each signal, as follows:

Term    Default action is to terminate the process.

Ign     Default action is to ignore the signal.

Core    Default action is to terminate the process and dump core (see core(5)).

Stop    Default action is to stop the process.

Cont    Default action is to continue the process if it is currently stopped.

Can adjust signal disposition with various system calls and establish **signal handlers** for the process.

# Standard Types of Signals

```
> man 7 signal
Standard Signals
```

Signal	x86 Value	Default Action	Comment
SIGHUP	1	Term	Hangup detected on controlling terminal or death of controlling process
SIGINT	2	Term	Interrupt from keyboard
SIGQUIT	3	Core	Quit from keyboard
SIGILL	4	Core	Illegal Instruction
SIGTRAP	5	Core	Trace/breakpoint trap
SIGABRT	6	Core	Abort signal from abort(3)
SIGBUS	7	Core	Bus error (bad memory access)
SIGFPE	8	Core	Floating-point exception (CK: actually integer divide by 0)
SIGKILL	9	Term	Kill signal
SIGUSR1	10	Term	User-defined signal 1
SIGSEGV	11	Core	Invalid memory reference
SIGUSR2	12	Term	User-defined signal 2
SIGPIPE	13	Term	Broken pipe: write to pipe with no readers; see pipe(7)
SIGALRM	14	Term	Timer signal from alarm(2)
SIGTERM	15	Term	Termination signal
SIGSTKFLT	16	Term	Stack fault on coprocessor (unused)
SIGCHLD	17	Ign	Child stopped or terminated
SIGCONT	18	Cont	Continue if stopped
SIGSTOP	19	Stop	Stop process
SIGTSTP	20	Stop	Stop typed at terminal
...			
SIGUNUSED	31	Core	Synonymous with SIGSYS

Note: Different CPU architectures may have different values for some signals and support other signals not listed

(Ex: MIPS CPUs use SIGCONT=25 with a synonym for SIGCHLD=19)

## Basic Signal Handlers via `signal()`

Pressing Ctrl-c in a terminal sends SIGINT to a running program which normally Terminates the program. The below template establishes a **signal handler** for SIGINT.

```
#include <signal.h>
void handle_SIGINT(int sig_num) {
    ...
}

int main () {
    // Set handling functions for programs
    signal(SIGINT, handle_SIGINT);
    ...
}
```

- ▶ When SIGINT arrives at program, control jumps to function `handle_SIGINT()` with argument `sig_num == SIGINT`
- ▶ When `handle_SIGINT()` completes, control returns to wherever the program left off

**Examine:** `no_interruptions_signal.c`

## Historical Notes 1 / 2

- ▶ Signals were an early concept but were initially “unreliable”: might get lost and so were not as useful as their modern incarnation

- ▶ Historically, required to reset signal handlers after they were called. First line of handler was always

```
signal(this_signal, this_handler);
```

though this was still buggy.

```
void handle_SIGINT(int sig_num) {  
    signal(SIGINT, handle_SIGINT); // Reset handler, unnecessary nowadays  
    printf("\nNo SIGINT-erruptions allowed.\n");  
    fflush(stdout);  
}  
  
int main () {  
    signal(SIGINT, handle_SIGINT); // Set handler the first time  
    ...  
}
```

- ▶ Old sources describe the need to reset handles while running
- ▶ What moment(s) in the program is no signal handler set?
- ▶ Not needed on *most* modern Unix systems

## Historical Notes 2 / 2

- ▶ Historically, some system calls could be interrupted by signals. Robbins & Robbins go on and on about this.

*On FreeBSD 8.0, Linux 3.2.0, and Mac OS X 10.6.8, when signal handlers are installed with the signal function, interrupted system calls will be restarted. The default on Solaris 10, however, is to return an error (EINTR) instead when system calls are interrupted by signal handlers installed with the signal function.*

*– Stevens and Rago, 10.5*

- ▶ Interrupted system calls meant EVERY system call had to appear in some sort of a try loop:

```
do {  
    ret = read(fd, buf, SIZE);           // read() once  
} while( ret == -1 && errno==EINTR ); // try again if interrupted  
                                     // completed a single read() call
```

- ▶ Modern `sigaction()` function does not have the problems of the old `signal()` function



# Portability Notes on `signal()`

```
> man 2 signal
```

```
...
```

The behavior of `signal()` varies across UNIX versions, and has also varied historically across different versions of Linux.

AVOID ITS USE: use `sigaction(2)` instead.

## PORTABILITY

The semantics when using `signal()` to establish a signal handler vary across systems (and POSIX.1 explicitly permits this variation);

*\*do not use it for this purpose.\**

- ▶ `signal()` part of the C standard but is old with different behaviors across different systems
- ▶ POSIX defined **new functions** which were designed to break from its tradition and fix problems associated with it

## Portable Signal Handlers via sigaction()

- ▶ The sigaction() function is more portable than signal() to register signal handlers.
- ▶ Makes use of struct sigaction which specifies properties of signal handler registrations, most importantly the field **sa\_handler**

```
int main(){                                // SAMPLE HANDLER SETUP USING sigaction()
    struct sigaction my_sa = {};           // portable signal handling setup with sigaction()
    my_sa.sa_handler = handle_signals;     // run function handle_signals
    sigemptyset(&my_sa.sa_mask);          // don't block any other signals during handling
    my_sa.sa_flags = SA_RESTART;           // restart system calls on signals if possible
    sigaction(SIGTERM, &my_sa, NULL);     // register SIGTERM with given action
    sigaction(SIGINT, &my_sa, NULL);      // register SIGINT with given action
    ...;
}
```

See no\_interruptions\_sigaction.c

# Ignoring Signals, Restoring Defaults

- ▶ Setting the signal handler to `SIG_IGN` will cause signals to be silently ignored.
- ▶ Setting the signal handler to `SIG_DFL` will restore default disposition.

Demo `no_interruptions_ignore.c`

# Sleeping, Pausing, and Stopping

## Sleeping/Pausing: wait for a signal

- ▶ `sleep(5)` suspends process execution until a signal is delivered or for 5 seconds elapses
- ▶ `pause()` suspends process execution until a signal is delivered;
- ▶ `sleep(0)` is equivalent to `pause()`

Note sleep behavior of various `no_interruptions` programs

## Signals that Affect Execution

- ▶ `SIGSTOP` will causes process to stop, will not resume until...
- ▶ `SIGCONT` causes a stopped process to resume, otherwise ignored by default
- ▶ All signals are delivered while a process is stopped BUT it is not resumed until receiving `SIGCONT`

**Examine:** `start_stop.c` with `circle_of_life.c`

# You want the Signal? You Can't Handle the Signal!

- ▶ SIGKILL and SIGSTOP cannot have dispositions changed from default
  - ▶ SIGKILL always terminates a process
  - ▶ SIGSTOP always stops a process execution
- ▶ In that sense they are a little different than the other signals but use the same OS delivery mechanism and `kill()` semantics
- ▶ Calls to `sigaction()` or `signal()` for these two will fail
- ▶ See `cant_handle_kill.c`

## Exercise: What Can you do with signals?

- ▶ Now have basics of signals and handlers in play
- ▶ Natural question: what are they good for?
- ▶ **Identify** some uses for signals that we have seen so far:
  - ▶ Standard uses for signals that have been demonstrated
  - ▶ How to use signals in this way
- ▶ **Propose** some uses for signals and handlers that are new and different from our examples so far

## Answers: What Can you *do* with signals?

- ▶ Kill programs via `kill(pid, SIGKILL)` or `kill -9 pid`
- ▶ Signals used in Shell for job control
  - ▶ Ctrl-Z suspends, uses `kill(pid, SIGSTOP)`
  - ▶ `fg / bg` resumes, uses `kill(pid, SIGCONT)`
- ▶ Catch `SIGTERM / SIGINT` and shut down gracefully:
  - ▶ Save files, close network connections, write to Databases etc.
  - ▶ Many programs do not want to suddenly die when in a sensitive state
  - ▶ Examples in Lab this week, used in P2 / P3
- ▶ Perform limited, dynamic responses to signals though this is tricky and there are usually better methods than signals that we will discuss

## Other Parts of struct sigaction

The struct sigaction argument to the sigaction() function allows several options for handlers to be specified.

Type	Field	Purpose
void(*) (int)	sa_handler	Pointer to a signal-catching function or one of the macros SIG_IGN or SIG_DFL.
sigset_t	sa_mask	Additional set of signals to be blocked during execution of signal-catching function.
int	sa_flags	Special flags to affect behavior of signal. Typically SA_RESTART is used to restart system calls automatically
void(*) (int, siginfo_t *, void *)	sa_sigaction	More complex handler used when sa_flags has SA_SIGINFO set; passes additional info to handler like PID of signaling process.

Standard setup for sigaction() call is

```
struct sigaction my_sa = {};  
sigemptyset(&my_sa.sa_mask);           // don't block other signals in ha  
my_sa.sa_flags = SA_RESTART;            // always restart system calls on  
my_sa.sa_handler = handle_SIGTERM;      // run function handle_SIGTERM  
sigaction(SIGTERM, &my_sa, NULL);      // register SIGTERM with given act
```



# Dangers in Signal Handlers

- ▶ General advice: do as little as possible in a signal handler
- ▶ Make use of only **reentrant** functions
  - ... reentrant if it can be interrupted in the middle of its execution, and then be safely called again (“re-entered”) before its previous invocations complete execution.*
  - [Wikipedia: Reentrancy](#)
- ▶ Notably NOT reentrant
  - `printf()` family, `malloc()`, `free()`
- ▶ Reentrant functions pertinent to thread-based programming as well (discussed later)

## Leading Example: `crypt_not_reentrant.c`

- ▶ Makes use of library call to `crypt()` which is used to generate encrypted versions of passwords
- ▶ `crypt()` called in both...
  - ▶ `main()` during a `while()` loop
  - ▶ in a signal handler for alarms
- ▶ `crypt()` is non-reentrant: **why?**
- ▶ Observe what happens during runs of program<sup>1</sup>

### Note: `alarm(secs)`

- ▶ Request to OS to send `SIGALRM` to program later on
- ▶ Alerts program that a certain amount of time has passed

---

<sup>1</sup>Similar example is in `getpwnam_not_reentrant.c`

## Exercise: Non-Reentrant Function Example

- ▶ Program calls non-reentrant function `f()` in two locations
  - ▶ `main()`
  - ▶ `handle_signal()` (!)
- ▶ With no signals, expect to see 7 printed
- ▶ If signaled should see 19, 7 printed in either order
- ▶ Show a control flow involving signals that prints 19 twice
- ▶ Why is `f()` not reentrant?

```
1 int z;  
2 int f(int x, int y){  
3     int tmp = x + y;  
4     z = tmp * 2 + 1;  
5     return z;  
6 }  
7  
8 void handle_signal(int sig){  
9     int t = f(4,5);  
10    printf("%d\n",t);  
11    return;  
12 }  
13  
14 int main(){  
15     signal(SIGINT,handle_signal);  
16     int v = f(1,2);  
17     printf("%d\n",v);  
18 }
```

## Answer: Non-Reentrant Function Example

- ▶ Program below calls non-reentrant function `f()` in both `main()` and `handle_signal()`
- ▶ With no interrupts, would expect to see 7 printed, with interrupts see 19 and 7
- ▶ Right hand shows one possible flow through the code which produces 19 then 19 again

```
1 int z;
2 int f(int x, int y){
3     int tmp = x + y;
4     z = tmp * 2 + 1;
5     return z;
6 }
7
8 void handle_signal(int sig){
9     int t = f(4,5);
10    printf("%d\n",t);
11    return;
12 }
13
14 int main(){
15     signal(SIGINT,handle_signal);
16     int v = f(1,2);
17     printf("%d\n",v);
18 }
```

EXECUTION STARTS IN `main()`

```
15: signal(SIGINT,handle_signal);
16: int v = f(1,2); // main(), Expect: (1+2)*2+1 = 7
   3: tmp = x + y;   // f(1,2): tmp = 1+2 = 3
   4: z = tmp*2 + 1; // z is 7
```

SIGINT delivered, run handler

```
9: int t = f(4,5); // handle_signal(2)
  3: tmp = x + y;   // f(4,5): tmp = 4+5 = 9
  4: z = tmp*2 + 1; // z is now 19
  5: return z;      // back to handle_signal()
  9: int t = f(4,5); // finished, t is 19
10: printf("%d\n",t); // PRINT 19
11: return;         // back to normal control
  5: return z;      // back to main(), but z is 19
16: int v = f(1,2); // v is actually 19
17: printf("%d\n",v); // PRINT 19
                        // 7 Expected
```

# Signal Sets

- ▶ A set of signals, likely implemented as a bit vector
- ▶ Functions allow addition, removal, clearing of set and tests for membership

```
#include <signal.h>

int sigemptyset(sigset_t *set);
// empty out the set

int sigfillset(sigset_t *set);
// fill the entire set with all signals

int sigaddset(sigset_t *set, int signo);
// add given signal to the set

int sigdelset(sigset_t *set, int signo);
// remove given signal to the set

// All of the above return 0 on succes, -1 on error

int sigismember(const sigset_t *set, int signo);
// return 1 if signal is a member of set, 0 if not
```

Examine sigsets\_demo.c

## Blocking (Disabling) Signals

- ▶ Processes can **block** signals, disable receiving them
- ▶ Signal is still there, just awaiting delivery
- ▶ Blocking is different from Ignoring a signal
  - ▶ Ignored signals are received and discarded
  - ▶ Blocked signals will be delivered after unblocking
- ▶ Can protect **Critical Sections** of code with by blocking if signals would screw it up

### Process Signal Mask

Example: block all signals that can be blocked

```
sigset_t block_all, defaults;  
sigfillset( &block_all );  
sigprocmask(SIG_SETMASK, &block_all, &defaults);  
// contains all  
// block all signals  
// save defaults
```

Examine `no_interruptions_block.c`

## Exercise: Protect Non-Reentrant Call

Examine the code for `crypt_not_reentrant.c` and modify it to use signal blocking to protect the **critical region** associated with calls to `crypt()`.

- ▶ Create a mask for all signals
- ▶ Block all signals prior to function call
- ▶ Unblock after returning
- ▶ Use code like below

```
sigset_t block_all, defaults;  
sigfillset( &block_all );  
sigprocmask(SIG_SETMASK, &block_all, &defaults);  
// contains all  
// block all signals  
// save defaults
```

**Note:** Be *very careful* where you unblock signal handling in `main()` to avoid errors: protect the **Critical Section**

*Code for `crypt_not_reentrant.c` on next slide*

## Exercise: Protect Non-Reentrant Call

```
1 // crypt_not_reentrant.c
2
3 #define PASSWORD "password123"
4 #define SALT "00"
5 char reference[128];
6 void alarm_handler(int signo) {
7     printf("in signal handler\n");
8     char *crypted = crypt(PASSWORD,SALT);
9     if( strcmp(reference, crypted, 128) != 0 ){
10         printf("MISMATCH: %s\n",crypted);
11         exit(1);
12     }
13     printf("HANDLER ENCRYPTED: %s\n",crypted);
14     printf("leaving signal handler\n");
15     alarm(1);
16 }
17
18
19 int main(void) {
20     struct sigaction my_sa = {
21         .sa_handler = alarm_handler,
22     };
23     sigaction(SIGALRM, &my_sa, NULL);
24     alarm(1);
25
26     printf("Repeatedly crypting '%s'\n",PASSWORD);
27
28     char *refcrypted = crypt(PASSWORD,SALT);
29     strcpy(reference,refcrypted,128);
30     printf("REFERENCE ENCRYPTED: %s\n",reference);
31
32     int successes = 0;
33     while(1){
34         char *crypted = crypt(PASSWORD,SALT);
35         if( strcmp(reference, crypted, 128) != 0 ){
36             printf("MISMATCH: %s\n",crypted);
37             exit(1);
38         }
39         successes++;
40         if(successes % 10000 == 0){
41             printf("%d successes so far\n",successes);
42         }
43     }
44     return 0;
45 }
```



# Answers: Protect Non-Reentrant Call

```
1 // crypt_protected.c
2 // SAME AS BEFORE
3 #define PASSWORD "password123"
4 #define SALT      "00"
5 char reference[128];
6 void alarm_handler(int signo) {
7     printf("in signal handler\n");
8     char *cryptd = crypt(PASSWORD,SALT);
9     if( strcmp(reference, cryptd, 128) != 0 ){
10         printf("MISMATCH: %s\n",cryptd);
11         exit(1);
12     }
13     printf("HANDLER ENCRYPTED: %s\n",cryptd);
14     printf("leaving signal handler\n");
15     alarm(1);
16 }
17
18 // BLOCK DURING CRITICAL REGION
19 int main(void) {
20     struct sigaction my_sa = {
21         .sa_handler = alarm_handler,
22     };
23     sigaction(SIGALRM, &my_sa, NULL);
24     alarm(1);
25
26     printf("Repeatedly crypting '%s'\n",PASSWORD);
27
28     char *refcryptd = crypt(PASSWORD,SALT);
29     strncpy(reference,refcryptd,128);
30     printf("REFERENCE ENCRYPTED: %s\n",reference);
31
32     int successes = 0;
33     while(1){
34         sigset_t block_all, defaults;
35         sigfillset( &block_all );
36         sigprocmask(SIG_SETMASK, &block_all, &defaults);
37         char *cryptd = crypt(PASSWORD,SALT);
38         if( strcmp(reference, cryptd, 128) != 0 ){
39             printf("MISMATCH: %s\n",cryptd);
40             exit(1);
41         }
42         sigprocmask(SIG_SETMASK, &defaults, NULL);
43         successes++;
44         if(successes % 10000 == 0){
45             printf("%d successes so far\n",successes);
46         }
47     }
48     return 0;
49 }
```

# Hardware Analogs to Signals

- ▶ Unix Signals are a **software** mechanism: happens via OS mechanisms in code
- ▶ Similar **hardware** mechanisms exist and deserve mention as some are related to software signals

## Example 1: Division by 0

- ▶ Processor ALU performs division
- ▶ Div by 0 generates an exceptional condition which transfers control to a hardware exception handler
- ▶ Typical CPU response is to jump to OS code
- ▶ OS sends a software signal to running program as SIGFPE

See `div0.c` and explain the output...

## Example 2: Alarms

Hardware timer expires → hardware signal → software signal

# Hardware Exceptions, Interrupts, Traps

Hardware features **electrical signals** that can cause control jumps. Definitions vary somewhat but two general types are common.

## Trap

- ▶ Generated by specific assembly instructions
- ▶ Div by 0 is a trap due to use of `idivX` instruction
- ▶ Jumps to handler indicated by CPU table
- ▶ Generated and handled **synchronously**

## Interrupt

- ▶ Generated by hardware devices like a disk drive often to indicate completion of operation
- ▶ Jumps CPU to interrupt handler so OS can react, move process waiting for file load from blocked to unblocked
- ▶ Major parts of OS kernel handle hardware via **asynchronous** interrupts

## Just to Muddy the Waters further...

- ▶ Modern system calls are made via `sysenter` (32-bit) and `syscall` (64-bit), BUT...
- ▶ In old-school 32-bit x86 assembly, making a system call was done via the *interrupt instruction*  
`int 0x80`    # trigger interrupt 128, handled by OS kernel
- ▶ Referred to as “trapping to the OS”
- ▶ Is this a...
  1. Trap?
  2. Interrupt?
  3. Another example of computer jargon that makes you want to change majors?

# Signal Take-Home

- ▶ Signals provide a simple way for programs to perform limited communications
- ▶ Can send signals via `kill` utility and system call `kill()`
- ▶ Programs respond to signals in a default manner (“signal disposition”) that can be changed and customized via handlers
- ▶ Can `sleep()` or `pause()` a program until a signal is received
- ▶ Can block signals if needed
- ▶ First example of **asynchronous** events in programs which introduces dangers associated with non-reentrant functions
- ▶ Signals not good for general purpose communication but are useful to convey simple events like “wake up already”