11 Signals

That's a signal, Jerry, that's a signal! [snaps his fingers again] Signal!

George Costanza (Seinfeld)

Signals have been a unix construct since the beginning. They are a convenient way to deliver low-priority information and for users to interact with their programs when no other form of interaction is available like using standard input. Signals allow a program to cleanup or perform an action in the case of an event. Some time, a program can choose to ignore events and that is completely fine and even supported by the standard. Crafting a program that uses signals well is tricky due to all the rules with inheritance. As such, signals are usually kept as cleanup or termination measures.

This chapter will go over how to first read information from a process that has either exited or been signaled and then it will deep dive into what are signals, how does the kernel deal with a signal, and the various ways processes can handle signals both in a single and multithreaded way.

The Deep Dive of Signals

A signal is a construct provided to us by the kernel. It allows one process to asynchronously send an event (think a message) to another process. If that process wants to accept the signal, it can, and then, for most signals, can decide what to do with that signal. Here is a short list (non comprehensive) of signals. The overall process for how a kernel sends a signal as well as common use cases are below.

1. Before any signals are generated, the kernel sets up the default signal handlers for a process.
2. If still no signals have arrived, the process can install its own signal handlers. This is simple telling the kernel that when the process gets signal X it should jump to function Y.
3. Now is the fun part, time to deliver a signal! Signals can come from various places below. The signal is now in what we call the generated state.
4. As soon as the signal starts to get delivered by the kernel, it is in the pending state.
5. The kernel then checks the signals `disposition`, which in layperson terms is whether the process is willing to accept that signal at this point. If not, then the signal is currently blocked and nothing happens.
6. If not, and there is no signal handler installed, the kernel executes the default action. Otherwise, the kernel delivers the signal by stopping *whatever* the process is doing at the current point, and jumping that process to the signal handler. If the program is multithreaded, then the process picks on thread with a signal disposition that can accept the signal and freezes the rest. The signal is now in the delivered phase.
7. Finally, we consider a signal caught if the process remains in tact after the signal was delivered.

Name	Default Action	Usual Use
SIGINT	Terminate (Can be caught)	Stop a process nicely
SIGQUIT	Terminate (Can be caught)	Stop a process harshly
SIGSTOP	Stop Process (Cannot be caught)	Suspends a process
SIGCONT	Continues a process	Starts after a stop
SIGKILL	Terminate Process (Cannot be caught)	You want the process gone

Sending Signals

Signals can be generated multiple ways. The user can send a signal. For example, you are at the terminal, and you send `CTRL-C` this is rarely the case in operating systems but is included in user programs for convenience. Another way is when a system event happens. For example, if you access a page that you aren't supposed to, the hardware generates a segfault interrupt which gets intercepted by the kernel. The kernel finds the process that caused this and sends a software interrupt signal `SIGSEGV`. There are softer kernel events like a child being created or sometimes when the kernel wants to like when it is scheduling processes. Finally, another process can send a message when you execute `kill -9 PID`, it sends `SIGKILL` to the process. This could be used in low-stakes communication of events between process. If you are relying on signals to be the driver in your program, you should rethink your application design. There are many drawbacks to using signals for asynchronous communication that is avoided by having a dedicated thread and some form of proper Interprocess Communication.

You can temporarily pause a running process by sending it a SIGSTOP signal. If it succeeds it will freeze a process, the process will not be allocated any more CPU time. To allow a process to resume execution send it the SIGCONT signal. For example, Here's program that slowly prints a dot every second, up to 59 dots.

```c
#include <unistd.h>
#include <stdio.h>
int main() {
  printf("My pid is %d\n", getpid() );
  int i = 60;
  while(--i) {
    write(1, ".",1);
    sleep(1);
  }
  write(1, "Done!",5);
  return 0;
}
```

We will first start the process in the background (notice the & at the end). Then send it a signal from the shell process by using the kill command.

```
>./program &
My pid is 403
...
>kill -SIGSTOP 403
>kill -SIGCONT 403
```

In C, you can send a signal to the child using `kill` POSIX call,

```
kill(child, SIGUSR1); // Send a user-defined signal
kill(child, SIGSTOP); // Stop the child process (the child cannot prevent this)
kill(child, SIGTERM); // Terminate the child process (the child can prevent this)
kill(child, SIGINT); // Equivalent to CTRL-C (by default closes the process)
```

As we saw above there is also a `kill` command available in the shell. Another command `killall` works the exact same way but instead of looking up by PID, it tries to match the name of the process. `ps` is an important utility that can help you find the pid of a process.

```
# First let's use ps and grep to find the process we want to send a signal to
$ ps au | grep myprogram
angrave  4409   0.0  0.0  2434892    512 s004  R+    2:42PM   0:00.00 myprogram 1 2 3

#Send SIGINT signal to process 4409 (equivalent of 'CTRL-C')
$ kill -SIGINT 4409

#Send SIGKILL (terminate the process)
$ kill -SIGKILL 4409
$ kill -9 4409
# Use kill all instead
$ killall -l firefox
```

In order to send a signal to the current, use `raise` or `kill` with `getpid()`

```
raise(int sig); // Send a signal to myself!
kill(getpid(), int sig); // Same as
```

For non-root processes, signals can only be sent to processes of the same user. You cant just SIGKILL my processes! `man -s2 kill` for more details.

Handling Signals

There are strict limitations on the executable code inside a signal handler. Most library and system calls are not `async-signal-safe` - they may not be used inside a signal handler because they are not re-entrant safe. Re-entrant safe means that imagine that your function can be frozen at any point and executed again, can you guarantee that your function wouldn't fail? Let's take the following

```
int func(const char *str) {
  static char buffer[200];
  strncpy(buffer, str, 199); # We finish this line and get recalled
  printf("%s\n", buffer)
}
```

1. We execute (func("Hello"))
2. The string gets copied over to the buffer completely (strcmp(buffer, "Hello") == 0)
3. A signal is delivered and the function state freezes, we also stop accepting any new signals until after the handler (we do this for convenience)
4. We execute `func("World")`
5. Now (strcmp(buffer, "World") == 0) and the buffer is printed out "World".
6. We resume the interrupted function and now print out the buffer once again "World" instead of what the function call originally intended "Hello"

Guaranteeing that your functions are signal handler safe are not as simple as not having shared buffers. You must also think about multithreading and synchronization i.e. what happens when I double lock a mutex? You also have to make sure that each function call is reentrant safe. Suppose your original program was interrupted while executing the library code of `malloc` ; the memory structures used by malloc will not be in a consistent state. Calling `printf` (which uses `malloc`) as part of the signal handler is unsafe and will result in `undefined behavior` i.e. it is no longer a useful,predictable program. In practice your program might crash, compute or generate incorrect results or stop functioning (`deadlock`),

depending on exactly what your program was executing when it was interrupted to execute the signal handler code. One common use of signal handlers is to set a boolean flag that is occasionally polled (read) as part of the normal running of the program. For example,

```
int pleaseStop ; // See notes on why "volatile sig_atomic_t" is better

void handle_sigint(int signal) {
  pleaseStop = 1;
}

int main() {
  signal(SIGINT, handle_sigint);
  pleaseStop = 0;
  while ( ! pleaseStop) {
     /* application logic here */
   }
  /* cleanup code here */
}
```

The above code might appear to be correct on paper. However, we need to provide a hint to the compiler and to the CPU core that will execute the `main()` loop. We need to prevent a compiler optimization: The expression `! pleaseStop` appears to be a loop invariant meaning it will be true forever, so can be simplified to `true`. Secondly, we need to ensure that the value of `pleaseStop` is not cached using a CPU register and instead always read from and written to main memory. The `sig_atomic_t` type implies that all the bits of the variable can be read or modified as an `atomic operation` - a single uninterruptible operation. It is impossible to read a value that is composed of some new bit values and old bit values.

By specifying `pleaseStop` with the correct type `volatile sig_atomic_t`, we can write portable code where the main loop will be exited after the signal handler returns. The `sig_atomic_t` type can be as large as an `int` on most modern platforms but on embedded systems can be as small as a `char` and only able to represent (-127 to 127) values.

```
volatile sig_atomic_t pleaseStop;
```

Two examples of this pattern can be found in `COMP` a terminal based 1Hz 4bit computer [3]. Two boolean flags are used. One to mark the delivery of `SIGINT` (CTRL-C), and gracefully shutdown the program, and the other to mark `SIGWINCH` signal to detect terminal resize and redraw the entire display.

You can also choose a handle pending signals asynchronously or synchronously. Install a signal handler to asynchronously handle signals use `sigaction` (or, for simple examples, `signal`). To synchronously catch a pending signal use `sigwait` which blocks until a signal is delivered or `signalfd` which also blocks and provides a file descriptor that can be `read()` to retrieve pending signals.

Sigaction

You should use `sigaction` instead of `signal` because it has better defined semantics. `signal` on different operating system does different things which is **bad** `sigaction` is more portable and is better defined for threads if need be. To change the `signal disposition` of a process - i.e. what happens when a signal is delivered to your process - use `sigaction` You can use system call `sigaction` to set the current handler for a signal or read the current signal handler for a particular signal.

```
int sigaction(int signum, const struct sigaction *act, struct sigaction
    *oldact);
```

The sigaction struct includes two callback functions (we will only look at the 'handler' version), a signal mask and a flags field -

```
struct sigaction {
              void     (*sa_handler)(int);
              void     (*sa_sigaction)(int, siginfo_t *, void *);
              sigset_t  sa_mask;
              int       sa_flags;
};
```

Suppose you installed a signal handler for the alarm signal,

```
signal(SIGALRM, myhandler);
```

The equivalent `sigaction` code is:

```
struct sigaction sa;
sa.sa_handler = myhandler;
sigemptyset(&sa.sa_mask);
sa.sa_flags = 0;
sigaction(SIGALRM, &sa, NULL)
```

However, we typically may also set the mask and the flags field. The mask is a temporary signal mask used during the signal handler execution. The `SA_RESTART` flag will automatically restart some (but not all) system calls that otherwise would have returned early (with EINTR error). The latter means we can simplify the rest of code somewhat because a restart loop may no longer be required.

```
sigfillset(&sa.sa_mask);
sa.sa_flags = SA_RESTART; /* Restart functions if interrupted by handler
    */
```

Sigwait

Sigwait can be used to read one pending signal at a time. `sigwait` is used to synchronously wait for signals, rather than handle them in a callback. A typical use of sigwait in a multi-threaded program is shown below. Notice that the thread signal mask is set first (and will be inherited by new threads). This prevents signals from being *delivered* so they will remain in a pending state until sigwait is called. Also notice the same set sigset_t variable is used by sigwait - except rather than setting the set of blocked signals it is being used as the set of signals that sigwait can catch and return.

One advantage of writing a custom signal handling thread (such as the example below) rather than a callback function is that you can now use many more C library and system functions that otherwise could not be safely used in a signal handler because they are not async signal-safe.

Based on Sigmask Code[2]

```
static sigset_t signal_mask; /* signals to block      */

int main (int argc, char *argv[]) {
```

```
    pthread_t sig_thr_id;    /* signal handler thread ID */
    sigemptyset (&signal_mask);
    sigaddset (&signal_mask, SIGINT);
    sigaddset (&signal_mask, SIGTERM);
    pthread_sigmask (SIG_BLOCK, &signal_mask, NULL);

    /* New threads will inherit this thread's mask */
    pthread_create (&sig_thr_id, NULL, signal_thread, NULL);

    /* APPLICATION CODE */
    ...
}

void *signal_thread (void *arg) {
    int      sig_caught;   /* signal caught     */

    /* Use same mask as the set of signals that we'd like to know about! */
    sigwait(&signal_mask, &sig_caught);
    switch (sig_caught)
    {
    case SIGINT:    /* process SIGINT */
        ...
        break;
    case SIGTERM:   /* process SIGTERM */
        ...
        break;
    default:        /* should normally not happen */
        fprintf (stderr, "\nUnexpected signal %d\n", sig_caught);
        break;
    }
}
```

Signal Disposition

For each process, each signal has a disposition which means what action will occur when a signal is delivered to the process. For example, the default disposition SIGINT is to terminate it. The signal disposition can be changed by calling signal() (which is simple but not portable as there are subtle variations in its implementation on different POSIX architectures and also not recommended for multi-threaded programs) or `sigaction` (discussed later). You can imagine the processes' disposition to all possible signals as a table of function pointers entries (one for each possible signal).

The default disposition for signals can be to ignore the signal, stop the process, continue a stopped process, terminate the process, or terminate the process and also dump a 'core' file. Note a core file is a representation of the processes' memory state that can be inspected using a debugger.

Multiple signals cannot be queued. However it is possible to have signals that are in a pending state. If a signal is pending, it means it has not yet been delivered to the process. The most common reason for a signal to be pending is that the process (or thread) has currently blocked that particular signal. If a particular signal, e.g. SIGINT, is pending then it is not possible to queue up the same signal again. It *is* possible to have more than one signal of a different type in a pending state. For example SIGINT and SIGTERM signals may be pending (i.e. not yet delivered to the target process)

Signals can be blocked (meaning they will stay in the pending state) by setting the process signal mask or, when you are writing a multithreaded program, the thread signal mask.

Disposition in Child Processes (No Threads)

After forking, The child process inherits a copy of the parent's signal dispositions and a copy of the parent's signal mask. In other words, if you have installed a SIGINT handler before forking, then the child process will also call the handler if a SIGINT is delivered to the child. Also if `SIGINT` is blocked in the parent, it will be blocked in the child too. Note pending

signals for the child are *not* inherited during forking. But after `exec`, both the signal mask and the signal disposition carries over to the exec-ed program [1]. Pending signals are preserved as well. Signal handlers are reset, because the original handler code has disappeared along with the old process.

To block signals use `sigprocmask`! With sigprocmask you can set the new mask, add new signals to be blocked to the process mask, and unblock currently blocked signals. You can also determine the existing mask (and use it for later) by passing in a non-null value for oldset.

```
int sigprocmask(int how, const sigset_t *set, sigset_t *oldset);‘
```

From the Linux man page of sigprocmask,

```
SIG_BLOCK: The set of blocked signals is the union of the current set and the
  set argument.
SIG_UNBLOCK: The signals in set are removed from the current set of blocked
  signals. It is permissible to attempt to unblock a signal which is not blocked.
SIG_SETMASK: The set of blocked signals is set to the argument set.
```

The sigset type behaves as a bitmap, except functions are used rather than explicitly setting and unsetting bits using & and |. It is a common error to forget to initialize the signal set before modifying one bit. For example,

```
sigset_t set, oldset;
sigaddset(&set, SIGINT); // Ooops!
sigprocmask(SIG_SETMASK, &set, &oldset)
```

Correct code initializes the set to be all on or all off. For example,

```
sigfillset(&set); // all signals
sigprocmask(SIG_SETMASK, &set, NULL); // Block all the signals!
// (Actually SIGKILL or SIGSTOP cannot be blocked...)

sigemptyset(&set); // no signals
sigprocmask(SIG_SETMASK, &set, NULL); // set the mask to be empty again
```

Signals in a multithreaded program

The new thread inherits a copy of the calling thread's mask. On initialization the calling thread's mask is the exact same as the processes mask because threads are essentially processes. After a new thread is created though, the processes signal mask turns into a gray area. Instead, the kernel likes to threat the process as a collection of thread, each of which can institute a signal mask and receive signals. In order to start setting your mask you can use,

```
pthread_sigmask( ... ); // set my mask to block delivery of some signals
pthread_create( ... ); // new thread will start with a copy of the same
    mask
```

Blocking signals is similar in multi-threaded programs to single-threaded programs: * Use `pthread_sigmask` instead of `sigprocmask` * Block a signal in all threads to prevent its asynchronous delivery

The easiest method to ensure a signal is blocked in all threads is to set the signal mask in the main thread before new threads are created

```
sigemptyset(&set);
sigaddset(&set, SIGQUIT);
sigaddset(&set, SIGINT);
pthread_sigmask(SIG_BLOCK, &set, NULL);

// this thread and the new thread will block SIGQUIT and SIGINT
pthread_create(&thread_id, NULL, myfunc, funcparam);
```

Just as we saw with `sigprocmask`, `pthread_sigmask` includes a 'how' parameter that defines how the signal set is to be used:

```
pthread_sigmask(SIG_SETMASK, &set, NULL) - replace the thread's mask with
    given signal set
pthread_sigmask(SIG_BLOCK, &set, NULL) - add the signal set to the
    thread's mask
pthread_sigmask(SIG_UNBLOCK, &set, NULL) - remove the signal set from the
    thread's mask
```

A signal then can delivered to any signal thread that is not blocking that signal. If the two or more threads can receive the signal then which thread will be interrupted is arbitrary! A common practice is to have one thread that can receive all signals or if there is a certain signal that requires special logic, have multiple threads for multiple signals. Even though programs from the outside can't send signals to specific threads (unless a thread is assigned a signal), you can do that in your program with `pthread_kill(pthread_t thread, int sig)`. In the example below, the newly created thread executing `func` will be interrupted by `SIGINT`

```
pthread_create(&tid, NULL, func, args);
pthread_kill(tid, SIGINT);
pthread_kill(pthread_self(), SIGKILL); // send SIGKILL to myself
```

As a word of warning `pthread_kill(threadid, SIGKILL)` will kill the entire process. Though individual threads can set a signal mask, the signal disposition (the table of handlers/action performed for each signal) is *per-process* not *per-thread*. This means `sigaction` can be called from any thread because you will be setting a signal handler for all threads in the process.

The linux man pages discusses signal system calls in section 2. There is also a longer article in section 7 (though not in OSX/BSD):

```
man -s7 signal
```

Topics

- Signals
- Signal Handler Safe

- Signal Disposition
- Signal States
- Pending Signals when Forking/Exec
- Signal Disposition when Forking/Exec
- Raising Signals in C
- Raising Signals in a multithreaded program

Questions

- What is a signal?
- How are signals served under UNIX? (Bonus: How about Windows?)
- What does it mean that a function is signal handler safe
- What is a process Signal Disposition?
- How do I change the signal disposition in a single threaded program? How about multithreaded?
- Why sigaction vs signal?
- How do I asynchronously and synchronously catch a signal?
- What happens to pending signals after I fork? Exec?
- What happens to my signal disposition after I fork? Exec?

Bibliography

[1] Executing a file. URL `https://www.gnu.org/software/libc/manual/html_node/Executing-a-File.html#Executing-a-File`.

[2] pthread$_s igmask.URL$.

Jure Åăorn. gto76/comp-cpp, Jun 2015. URL `https://github.com/gto76/comp-cpp/blob/1bf9a77eaf8f57f7358a316e5bba src/output.c`.