IC221: SYSTEMS PROGRAMMING (SP16)

Home Policy Calendar Resources

# LEC 19: SIGNALS AND SIGNAL HANDLING

## 1 What are signals and how are they used

A **signal** is a software interrupt, a way to communicate information to a process about the state of other processes, the operating system, and hardware. A signal is an **interrupt** in the sense that it can change the flow of the program —when a signal is delivered to a process, the process will stop what its doing, either handle or ignore the signal, or in some cases terminate, depending on the signal.

Signals may also be delivered in an unpredictable way, out of sequence with the program due to the fact that signals may originate outside of the currently executing process. Another way to view signals is that it is a mechanism for handling **asynchronous events**. As opposed to **synchronous events**, which is when a standard program executes iterative, that is, one line of code following another. **Asynchronous events** occur when portions of the program execute out of order. Asynchronous events typically occur due to external events originating at the hardware or operating system; the signal, itself, is the way for the operating system to communicate these events to the processes so that the process can take appropriate action.

### 1.1 How we use signals

Signals are used for a wide variety of purposes in Unix programming, and we've already used them in smaller contexts. For example, when we are working in the shell and wish to "kill all cat programs" we type the command:

```
#> killall cat
```

The `killall` command will send a signal to all processes named `cat` that says "terminate." The actually signal being sent is `SIGTERM`, whose purposes is to communicate a termination request to a given process, but the process does not actually have to terminate … more on that later.

We've also used and looked at signals in the context of **terminal signaling** which is how programs stop, start and terminate. When we type `Ctrl-c` that is the same as sending a `SIGINT` signal, and when we type `Ctrl-z` that is the same as sending a `SIGTSTP` signal, and when we type `fg` or `bg` that is the same as sending a `SIGCONT` signal.

Each of these signals describe an action that the process should take in response. This action is outside the normal control flow of the program, the events arrive **asynchronously** requiring the process to **interrupt** its current operations to respond to the event. For above signals, the response is clear — `SIGTERM` terminate, `SIGSTOP` stop, `SIGCONT` continue — but for other signals, the programmer can choose the correct response, which may be to simply ignore the signal all together.

## 2 The Wide World of Signals

Every signal has a name, it starts with `SIG` and ends with a description. We can view all the signals in section 7 of the man pages, below are the standard Linux signals you're likely to interact with:

```
Signal     Value     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
SIGABRT       6       Core    Abort signal from abort(3)
SIGFPE        8       Core    Floating point exception
SIGKILL       9       Term    Kill signal
SIGSEGV      11       Core    Invalid memory reference
SIGPIPE      13       Term    Broken pipe: write to pipe with no
                              readers
SIGALRM      14       Term    Timer signal from alarm(2)
SIGTERM      15       Term    Termination signal
SIGUSR1   30,10,16    Term    User-defined signal 1
SIGUSR2   31,12,17    Term    User-defined signal 2
SIGCHLD   20,17,18    Ign     Child stopped or terminated
SIGCONT   19,18,25    Cont    Continue if stopped
SIGSTOP   17,19,23    Stop    Stop process
SIGTSTP   18,20,24    Stop    Stop typed at tty
SIGTTIN   21,21,26    Stop    tty input for background process
SIGTTOU   22,22,27    Stop    tty output for background process
```

### 2.1 Signal Names and Values

Notice that each signal has a name, value, and default action. The signal name should start to become a bit more familiar, the value of a signal is actually the same as the signal itself. In fact, the signal name is just a `#defined` value, and we can see this by looking at the `sys/signal.h` header file:

```
#define SIGHUP  1       /* hangup */
#define SIGINT  2       /* interrupt */
#define SIGQUIT 3       /* quit */
#define SIGILL  4       /* illegal instruction (not reset when caught) */
#define SIGTRAP 5       /* trace trap (not reset when caught) */
#define SIGABRT 6       /* abort() */
#define SIGPOLL 7       /* pollable event ([XSR] generated, not supported) */
#define SIGFPE  8       /* floating point exception */
#define SIGKILL 9       /* kill (cannot be caught or ignored) */
//(...)
```

In code we use both the `#defined` value and the number. In general, it is easier to remember the name of the signal, but some signals are often referred to by value, in particular, `SIGKILL`, whose value 9 is affectionately used in the phrase: "Kill 9 that process."

### 2.2 Default Actions of Signals

Each signal has a default action. There are four described in the table:

- `Term` : The process will terminate
- `Core` : The process will terminate and produce a core dump file that traces the process state at the time of termination.
- `Ign` : The process will ignore the signal
- `Stop` : The process will stop, like with a `Ctrl-Z`
- `Cont` : The process will continue from being stopped

As we willsee later, for some signals, we can change the default actions. A few signals, which are control signals, cannot have their default action changed, these include `SIGKILL` and `SIGABRT`, which is why "kill 9" is the ultimate kill statement.

## 3 Signals from the Command Line

Terminology for delivering signals is to "kill" a process with the `kill` command. The `kill` command is actually poorly named — originally, it was only used to kill or terminate a process, but it is currently used to send any kind of signal to a process. The difference between `kill` and `killall` is that `kill` only sends signals to process identified by their pid, `killall` sends the signal to all process of a given name.

### 3.1 Preparing for the kill

A good exercise to explore the variety of signals and how to use them is to actually use them from the command line. To start, we can open **two** terminals, in one, we execute the loop program:

```
/*loop.c*/
int main(){ while(1); }
```

Which will just loop forever, and in other terminal, we will `kill` this process with various signals to see how it responds. Let's start with a signal we all love to hate, the signal that indicates a Segmentation Fault occurs. You might not have realized this, but a Segmentation Fault is a signal generated from the O.S. that something bad has happened. Let's simulate that effect:

```
killall -SIGSEGV loop
```

And in the in the terminal where `loop` is running, the result is eerily familiar.

```
#>./loop
Segmentation fault: 11
```

The 11 following the message is the signal number: 11 is the signal number for `SIGSGV`. Note that the default response to a `SIGSEGV` is to terminate with a core dump.

We can explore some of the more esoteric signals and see similar results occur when the program terminates:

```
| Signal     | Output                      |
|------------+-----------------------------|
| SIGKILL    | Killed: 9                   |
| SIGQUIT    | Quit: 3                     |
| SIGILL     | Illegal instruction: 4      |
| SIGABRT    | Abort trap: 6               |
| SIGFPE     | Floating point exception: 8 |
| SIGPIPE    | (no output)                 |
| SIGALAR    | Alarm clock: 14             |
| SIGUSR1    | User defined signal 1: 30   |
| SIGUSR2    | User defined signal 2: 31   |
|------------+-----------------------------|
```

### 3.2 Sending Terminal Signals with Kill

Let's restart the loop program and use `kill` to simulate terminal signaling. We've been discussing how the terminal control will deliver signals to stop, continue, and terminate a process; there's no mystery here. Those signals are signals that you can send yourself with `kill`.

Let's look at starting the loop program again, but this time

```
killall -SIGSTOP loop
```

And again, the result in the other terminal is quite familiar:

```
#>./loop

[1]+  Stopped                 ./loop
```

If we were to run `jobs`, we can see that `loop` is stopped in the background. This is the same as typing `Ctrl-z` in the terminal.

```
#> jobs
[1]+  Stopped                 ./loop
```

Before, we'd continue the loop program with a call to `bg` or `fg`, but we can use `kill` to do that too. From the other terminal:

```
killall -SIGCONT loop
```

And, after we run jobs, the `loop` program is running in the background:

```
#> jobs
[1]+  Running                 ./loop &
```

Finally, let's terminate the loop program. The `Ctrl-c` from the terminal actually generates the `SIGINT` signal, which stands for "interrupt" because a `Ctrl-c` initiates an interrupt of the foreground process, which by default terminates the process.

```
killall -SIGINT loop
```

And the expected result:

```
#> jobs
[1]+  Interrupt: 2            ./loop &
```

## 4 Handling and Generating Signals

Now that we have a decent understanding of signals and how they communicate information to a process, let's move on to investigate how we can write program that take some action based on a signal. This is described as **signal handling**, a program that handles a signal, either by ignoring it or taking some action when the signal is delivered. We will also explore how signals can be sent from one program to another, again, we'll use a `kill` for that.

### 4.1 Hello world of Signal Handling

The primary system call for signal handling is `signal()`, which given a signal and function, will execute the function whenever the signal is delivered. This function is called the **signal handler** because it *handles* the signal. The `signal()` function has a strange declaration:

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

That is, signal takes two arguments: the first argument is the signal number, such as `SIGSTOP` or `SIGINT`, and the second is a reference to a handler function whose first argument is an int and returns void. It's probably best to explore `signal()` through an example, and hello world program is where we always start.

```
#include <stdlib.h>
#include <stdio.h>

#include <signal.h> /*for signal() and raise()*/

void hello(int signum){
  printf("Hello World!\n");
}

int main(){

  //execute hello() when receiving signal SIGUSR1
  signal(SIGUSR1, hello);

  //send SIGUSR1 to the calling process
  raise(SIGUSR1);
}
```

The above program first establishes a signal handler for the user signal `SIGUSR1`. The signal handling function `hello()` does as expected: prints "Hello World!" to stdout. The program then sends itself the `SIGUSR1` signal, which is accomplished via `raise()`, and the result of executing the program is the beautiful phrase:

```
#> ./hello_signal
Hello World!
```

## 4.2 Asynchronous Execution

Some key points to take away from the hello program is that the second argument to `signal()` is a function pointer, a reference to a function to call. This tells the operating system that whenever this signal is sent to this process, run this function as the signal handler.

Also, the execution of the signal handler is asynchronous, which means the current state of the program will be paused while the signal handler executes, and then execution will resume from the pause point, much like context switching.

Let's look at another example hello world program:

```
/* hello_loop.c*/
void hello(int signum){
  printf("Hello World!\n");
}

int main(){

  //Handle SIGINT with hello
  signal(SIGINT, hello);

  //loop forever!
  while(1);

}
```

The above program will set a signal handler for `SIGINT` the signal that is generated when you type `Ctrl-C`. The question is, when we execute this program, what will happen when we type `Ctrl-C`?

To start, let's consider the execution of the program. It will register the signal handler and then will enter the infinite loop. When we hit `Ctrl-C`, we can all agree that the signal handler `hello()` should execute and "Hello World!" prints to the screen, but the program was in an infinite loop. In order to print "Hello World!" it must have been the case that it broke the loop to execute the signal handler, right? So it should exit the loop as well as the program. Let's see:

```
#> ./hello_loop
^CHello World!
^CHello World!
^CHello World!
^CHello World!
^CHello World!
^CHello World!
^CHello World!
^\Quit: 3
```

As the output indicates, every time we issued `Ctrl-C` "Hello World!" prints, but the program returns to the infinite loop. It is only after issuing a `SIGQUIT` signal with `Ctrl-\` did the program actually exit.

While the interpretation that the loop would exit is reasonable, it doesn't consider the primary reason for signal handling, that is, asynchronous event handling. That means the signal handler acts out of the standard flow of the control of the program; in fact, the whole program is saved within a context, and a new context is created just for the signal handler to execute in. If you think about it some more, you realize that this is pretty cool, and also a totally new way to view programming.

## 4.3 Inter Process Communication

Signals are also a key means for inter-process communication. One process can send a signal to another indicating that an action should be taken. To send a signal to a particular process, we use the `kill()` system call. The function declaration is below.

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

Much like the command line version, `kill()` takes a process identifier and a signal, in this case the signal value as an `int`, but the value is `#defined` so you can use the name. Let's see it in use.

```
/*ipc_signal.c*/
void hello(){
  printf("Hello World!\n");
}

int main(){

  pid_t cpid;
  pid_t ppid;

  //set handler for SIGUSR1 to hello()
  signal(SIGUSR1, hello);

  if ( (cpid = fork()) == 0){
    /*CHILD*/

    //get parent's pid
    ppid = getppid();

    //send SIGUSR1 signal to parrent
    kill(ppid, SIGUSR1);
    exit(0);

  }else{
    /*PARENT*/

    //just wait for child to terminate
    wait(NULL);
  }

}
```

In this program, first a signal handler is established for `SIGUSR1`, the `hello()` function. After the fork, the parent calls `wait()` and the child will communicate to the parent by "killing" it with the `SIGUSR1` signal. The result is that the handler is invoked in the parent and "Hello World!" is printed to stdout from the parent.

While this is a small example, signals are integral to inter process communication. In previous lessons, we've discussed how to communicate data between process with `pipe()`, signals is the way process communicate state changes and other asynchronous events. Perhaps most relevant is state change in child processes. The `SIGCHLD` signal is the signal that gets delivered to the parent when a child terminates. So far, we've been handling this signal implicitly through `wait()`, but you can choose instead to handle `SIGCHLD` and take different actions when a child terminates. We'll look at that in more detail in a future lesson.

## 4.4 Ignoring Signals

So far, our handlers have been doing things — mostly, printing "Hello World!" — but we might just want our handler to do nothing, essentially, ignoring the signal. That is easy enough to write in code, for example, here is a program that will ignore `SIGINT` by handling the signal and do nothing:

```
/*ingore_sigint.c*/
#include <signal.h>
#include <sys/signal.h>

void nothing(int signum){ /*DO NOTHING*/ }

int main(){

  signal(SIGINT, nothing);

  while(1);
}
```

And if we run this program, we see that, yes, it `Ctrl-c` is ineffective and we have to use `Ctrl-\` to quit the program:

```
>./ignore_sigint
^C^C^C^C^C^C^C^C^C^C^\Quit: 3
```

But, it would seem like a pain to always have to write the silly little ignore function that does nothing, and so, when there is a need, there is a way. The `signal.h` header defines a set of actions that can be used in place of the handler:

- `SIG_IGN` : Ignore the signal
- `SIG_DFL` : Replace the current signal handler with the default handler

With these keywords, we can rewrite the program simply as:

```
int main(){

  // using SIG_IGN
  signal(SIGINT, SIG_IGN);

  while(1);
}
```

## 4.5 Changing and Reverting to the default handler

Setting a signal handler is not a singular event. You can always change the handler and you can also revert the handler back to default state. For example, consider the following program:

```
/*you_shot_me.c*/
void handler_3(int signum){
  printf("Don't you dare shoot me one more time!\n");

  //Revert to default handler, will exit on next SIGINT
  signal(SIGINT, SIG_DFL);
}

void handler_2(int signum){
  printf("Hey, you shot me again!\n");

  //switch handler to handler_3
  signal(SIGINT, handler_3);
}

void handler_1(int signum){
  printf("You shot me!\n");

  //switch handler to handler_2
  signal(SIGINT, handler_2);
}


int main(){

  //Handle SIGINT with handler_1
  signal(SIGINT, handler_1);

  //loop forever!
  while(1);

}
```

The program first initiates `handler_1()` as the signal handler for `SIGINT`. After the first `Ctrl-c`, in the signal handler, the handler is changed to `handler_2()`, and after the second `Ctrl-c`, it is change again to `handler_3()` from `handler_2()`. Finally, in `handler_3()` the default signal handler is reestablished, which is to terminate on `SIGINT`, and that is what we see in the output:

```
#> ./you_shout_me
^CYou shot me!
^CHey, you shot me again!
^CDon't you dare shoot me one more time!
^C
```

## 4.6 Some signals are more equal than others

The last note on signal handling is that not all signals are created equal, and some signals are more equal than others. That means, that you cannot handle all signals because it could potentially place the system in an unrecoverable state.

The two signals that can never be ignored or handled are: `SIGKILL` and `SIGSTOP`. Let's look at an example:

```
/* ignore_stop.c */
int main(){

  //ignore SIGSTOP ?
  signal(SIGSTOP, SIG_IGN);

  //infinite loop
  while(1);

}
```

The above program tries to set the ignore signal handler for `SIGSTOP`, and then goes into an infinite loop. If we execute the program, we find that oure efforts were fruitless:

```
#>./ignore_stop
^Z
[1]+  Stopped                 ./ignore_stop
```

The program did stop. And we can see the same for a program that ignores `SIGKILL`.

```
int main(){

  //ignore SIGSTOP ?
  signal(SIGKILL, SIG_IGN);

  //infinite loop
  while(1);

}
```

```
#>./ignore_kill &
[1] 13129
#>kill -SIGKILL 13129
[1]+  Killed: 9               ./ignore_kill
```

The reasons for this are clearer when you consider that all programs must have a way to stop and terminate. These processes cannot be interfered with otherwise operating system would loose control of execution traces.

## 4.7 Checking Errors of signal()

The `signal()` function returns a pointer to the previous signal handler, which means that here, again, is a system call that we cannot error check in the typical way, by checking if the return value is less than 0. This is because a pointer type is unsigned, there is no such thing as negative pointers.

Instead, a special value is used `SIG_ERR` which we can compare the return value of `signal()`. Here, again, is the program where we try and ignore `SIGKILL`, but this time with proper error checking:

```
/*signal_errorcheck.c*/
int main(){

  //ignore SIGSTOP ?
  if( signal(SIGKILL, SIG_IGN) == SIG_ERR){
    perror("signal");;
    exit(1);
  }

  //infinite loop
  while(1);

}
```

And the output from the `perror()` is clear:

```
#>./signal_errorcheck
signal: Invalid argument
```

The invalid argument is `SIGKILL` which cannot be handled or ignored. It can only KILL!