Lecture_5_Signals_and_return_values

February 5, 2025

1 Introduction

Today, we'll dive into the core principles of signals in C:

- Understanding how signals work and their role in process management.
- Understanding how to send, handle, and block signals using standard signal functions.
- How signals enable asynchronous communication between processes.

Understanding signals in C is crucial because they allow programs to handle asynchronous events, manage processes, and respond to system interrupts efficiently.

---

2 Signals

C, as a language, does not inherently define signals but provides a standard library interface (via `<signal.h>`) to work with OS-level signals. Signals in C are a standardized subset of the broader Linux signal system. The C standard defines a subset of signals, including `SIGABRT`, `SIGFPE`, `SIGILL`, `SIGINT`, `SIGSEGV`, and `SIGTERM`.

Signals are software interrupts that provide a mechanism for handling asynchronous events. These events can originate from outside the system, such as when the user generates the interrupt character by pressing `Ctrl-C`, or from activities within the program or kernel, such as when the process executes code that divides by zero. As a primitive form of interprocess communication (IPC), one process can also send a signal to another process.

The key point is not just that the events occur asynchronously — the user, for example, can press `Ctrl-C` at any point in the program's execution — but also that the program handles the signals asynchronously. The signal-handling functions are registered with the kernel, which invokes the functions asynchronously from the rest of the program when the signals are delivered.

Signals have been part of Unix since the early days. Over time, however, they have evolved, most noticeably in terms of reliability, as signals once could get lost, and in terms of functionality, as signals may now carry user-defined payloads. At first, different Unix systems made incompatible changes to signals. Thankfully, POSIX came to the rescue and standardized signal handling. This standard is what Linux provides.

Most nontrivial applications interact with signals. Even if you deliberately design your application to not rely on signals for its communication needs — often a good idea! — you'll still be forced to work with signals in certain cases, such as when handling program termination.

3 Signal Concepts

Signals have a very precise lifecycle. First, a signal is `raised` (we sometimes also say it is *sent* or *generated*). The kernel then stores the signal until it is able to deliver it. Finally, once it is free to do so, the kernel handles the signal as appropriate. The kernel can perform one of three actions, depending on what the process asked it to do:

3.0.1 Ignore the signal

No action is taken. There are two signals that cannot be ignored: `SIGKILL` and `SIGSTOP`. The reason for this is that the system administrator needs to be able to kill or stop processes, and it would be a circumvention of that right if a process could elect to ignore a `SIGKILL` (making it unkillable) or a `SIGSTOP` (making it unstoppable).

3.0.2 Catch and handle the signal

The kernel will suspend execution of the process's current code path and jump to a previously registered function. The process will then execute this function. Once the process returns from this function, it will jump back to wherever it was when it caught the signal. `SIGINT` and `SIGTERM` are two commonly caught signals. Processes catch `SIGINT` to handle the user generating the interrupt character — for example, a terminal might catch this signal and return to the main prompt. Processes catch `SIGTERM` to perform necessary cleanup, such as disconnecting from the network or removing temporary files, before terminating. `SIGKILL` and `SIGSTOP` cannot be caught.

3.0.3 Perform the default action

This action depends on the signal being sent. The default action is often to terminate the process. This is the case with `SIGKILL`, for instance. However, many signals are provided for specific purposes that concern programmers in particular situations, and these signals are ignored by default because many programs are not interested in them. We will look at the various signals and their default actions shortly.

Traditionally, when a signal was delivered, the function that handled the signal had no information about what had happened except for the fact that the particular signal had occurred. Nowadays, the kernel can provide a lot of context to programmers who wish to receive it. Signals can even pass user-defined data.

4 Signal Identifiers

Every signal has a symbolic name that starts with the prefix `SIG`. For example, `SIGINT` is the signal sent when the user presses `Ctrl-C`, `SIGABRT` is the signal sent when the process calls the `abort()` function, and `SIGKILL` is the signal sent when a process is forcefully terminated.

These signals are all defined in a header file included from `<signal.h>`. The signals are simply preprocessor definitions that represent positive integers — that is, every signal is also associated with an integer identifier. The name-to-integer mapping for the signals is implementation-dependent and varies among Unix systems, although the first dozen or so signals are usually mapped the same way (`SIGKILL` is infamously signal `9` for example). A portable program will always use a signal's human-readable name, and never its integer value.

The signal numbers start at `1` (generally `SIGHUP`) and proceed linearly upward. There are about `31` signals in total, but most programs deal regularly with only a handful of them. There is no signal with the value `0`, which is a special value known as the `null` signal. There's really nothing important about the `null` signal — it doesn't deserve a special name — but some system calls (such as `kill()`) use a value of `0` as a special case.

> You can generate a list of signals supported on your system with the command `kill -l`

The table listing the signals that Linux supports, along with their descriptions and default actions:

Signal	Description	Default Action
SIGHUP	Hangup detected on controlling terminal	Terminate
SIGINT	Interrupt from keyboard (Ctrl+C)	Terminate
SIGQUIT	Quit from keyboard (Ctrl+)	Core dump
SIGILL	Illegal instruction	Core dump
SIGABRT	Abnormal termination (abort)	Core dump
SIGFPE	Floating-point exception	Core dump
SIGKILL	Kill signal (cannot be caught or ignored)	Terminate
SIGSEGV	Segmentation fault (invalid memory access)	Core dump
SIGPIPE	Broken pipe (write to a closed pipe)	Terminate
SIGALRM	Alarm clock signal	Terminate
SIGTERM	Termination signal	Terminate
SIGUSR1	User-defined signal 1	Terminate
SIGUSR2	User-defined signal 2	Terminate
SIGCHLD	Child process terminated, stopped, or resumed	Ignore
SIGCONT	Continue if stopped	Continue
SIGSTOP	Stop process (cannot be caught or ignored)	Stop
SIGTSTP	Stop typed at terminal (Ctrl+Z)	Stop
SIGTTIN	Background process attempts read from terminal	Stop
SIGTTOU	Background process attempts write to terminal	Stop
SIGBUS	Bus error (bad memory access)	Core dump
SIGPOLL	Pollable event (Sys V signal)	Terminate
SIGPROF	Profiling timer expired	Terminate
SIGSYS	Bad system call	Core dump
SIGTRAP	Trace/breakpoint trap	Core dump
SIGURG	Urgent condition on socket	Ignore
SIGVTALRM	Virtual alarm clock	Terminate
SIGXCPU	CPU time limit exceeded	Core dump
SIGXFSZ	File size limit exceeded	Core dump
SIGWINCH	Window resize signal	Ignore
SIGIO	I/O now possible	Terminate
SIGPWR	Power failure/restart	Terminate
SIGSTKFLT	Stack fault on coprocessor	Terminate
SIGRTMIN	First real-time signal	Varies
SIGRTMAX	Last real-time signal	Varies

5 Basic Signal Management

With the signals out of the way, we'll now turn to how you manage them from within your program. The simplest and oldest interface for signal management is the `signal()` function. Defined by the `ISO C89` standard, which standardizes only the lowest common denominator of signal support, this system call is very basic. Linux offers substantially more control over signals via other interfaces. Because `signal()` is the most basic and, thanks to its presence in `ISO C`, quite common.

5.1 Example 1: Handling Signals with `signal()`

`signal()` takes two parameters: * The signal number for which the handler is being set (e.g. `SIGINT`, `SIGTERM`, `SIGALRM` etc.) * A pointer to the signal handler function.

The `signal()` function returns the previous handler for the signal, or `SIG_ERR` if an error occurs.

[ ]:
```
%%writefile signal.c

#include <stdio.h>
#include <signal.h>
#include <unistd.h>
#include <stdlib.h>

// Signal handler function for SIGINT
void handle_sigint(int sig) {
    printf("\nCaught SIGINT (Signal number: %d). Exiting...\n", sig);
    exit(0); // Exit the program after handling the signal
}

int main() {
    // Register the signal handler for SIGINT
    signal(SIGINT, handle_sigint);

    // Print instructions once
    printf("Running... Press Ctrl+C to exit.\n");
    printf("In Google Colab, press Ctrl+M I to interrupt.\n");

    // Infinite loop to keep the program running
    while (1) {
        printf("Running...\n");
        sleep(1);
    }

    return 0;
}
```

```
Overwriting signal.c
```

[ ]:
```
%%script bash
gcc signal.c -o signal
```

[ ]:
```
!./signal
```

```
Running… Press Ctrl+C to exit.
In Google Colab, press Ctrl+M I to interrupt.
Running…
Running…
Running…
Running…
Running…
Running…

Caught SIGINT (Signal number: 2). Exiting…
```

The `exit` function is declared in `<stdlib.h>`, so this header file must be included to use `exit`.

When we run the program, it prints the instructions once and then enters an infinite loop, printing "`Working...`" every second.

In Google Colab, the terminal interrupt key combination is `Ctrl+M I` instead of `Ctrl+C`. This is why the message specifically mentions Colab.

If you press `Ctrl+C` (or `Ctrl+M I` in Google Colab), the `SIGINT` signal is sent to the program, and the `handle_sigint` function is executed.

The program prints "`Caught SIGINT (Signal number: 2). Exiting...`" and then terminates.

5.2 Example 2: Ignoring a Signal

[ ]:
```
%%writefile ignor.c

#include <stdio.h>
#include <signal.h>
#include <unistd.h>

int main() {
    signal(SIGINT, SIG_IGN);  // SIG_IGN tells the system to ignore SIGINT.

    while (1) {
        printf("SIGINT (Ctrl+C) is ignored. Press Ctrl+Z to stop.\n");
        sleep(2);
    }

    return 0;
}
```

```
Overwriting ignor.c
```

[ ]:
```
%%script bash
gcc ignor.c -o ignor
```

[ ]:
```
!./ignor
```

```
SIGINT (Ctrl+C) is ignored. Press Ctrl+Z to stop.
SIGINT (Ctrl+C) is ignored. Press Ctrl+Z to stop.
SIGINT (Ctrl+C) is ignored. Press Ctrl+Z to stop.
SIGINT (Ctrl+C) is ignored. Press Ctrl+Z to stop.
SIGINT (Ctrl+C) is ignored. Press Ctrl+Z to stop.
^C
```

5.3 Example 3: Handling Signals with `signal()`

[ ]:
```c
%%writefile sigaction.c

#include <stdio.h>
#include <signal.h>
#include <unistd.h>
#include <stdlib.h>
#include <string.h>

// Signal handler function for SIGINT
void handle_sigint(int sig, siginfo_t *info, void *context) {
    printf("\nCaught SIGINT (Signal number: %d). Exiting...\n", sig);
    printf("Sender PID: %d, User ID: %d\n", info->si_pid, info->si_uid);
    exit(0); // Exit the program after handling the signal
}

int main() {
    struct sigaction sa;

    // Clear the structure
    memset(&sa, 0, sizeof(sa));

    // Set the handler function
    sa.sa_sigaction = handle_sigint;

    // Use the SA_SIGINFO flag to get additional information
    sa.sa_flags = SA_SIGINFO;

    // Register the signal handler for SIGINT
    if (sigaction(SIGINT, &sa, NULL) == -1) {
        perror("sigaction");
        exit(EXIT_FAILURE);
    }

    // Print instructions
    printf("Running... Press Ctrl+C to exit.\n");
    printf("In Google Colab, press Ctrl+M I to interrupt.\n");
    // Infinite loop to keep the program running
    while (1) {
        printf("Working...\n");
        sleep(1);
    }

    return 0;
}
```

```
Overwriting sigaction.c
```

[ ]:
```
%%script bash
gcc sigaction.c -o sigaction
```

[ ]:
```
!./sigaction
```

```
Running… Press Ctrl+C to exit.
In Google Colab, press Ctrl+M I to interrupt.
Working…
Working…
Working…
Working…
Working…
Working…
Working…
Working…
Working…
Working…
Working…
Working…
Working…
Working…
Working…
Working…
Working…
Working…

Caught SIGINT (Signal number: 2). Exiting…
Sender PID: 92, User ID: 0
```

`struct sigaction` is used to specify the behavior of the signal handler. `sa.sa_sigaction` is set to the handler function (`handle_sigint`).

The `handle_sigint` receives the signal number (`sig`), a pointer to a `siginfo_t` structure (`info`), and a `void` pointer (`context`). The `siginfo_t` structure contains additional information, such as the sender's PID (`si_pid`) and user ID (`si_uid`).

The `sigaction(SIGINT, &sa, NULL)` call registers the signal handler for `SIGINT`. If the registration fails, `perror` is used to print an error message, and the program exits with a failure status.

The program prints “`Working...`” every second to indicate that it is running. The loop continues until the user interrupts the program with `Ctrl+C` (or `Ctrl+M I` in Google Colab).

When the signal is received, the handler prints a message, including the sender’s PID and user ID, and then exits the program using `exit(0)`.