# CS232L Operating Systems Lab
# Lab 13: Using Signals to communicate between processes

CS Program
Habib University

Fall 2021

## 1 Introduction

Signals is another mechanism to communicate between two running processes on a unix-like operating system.

In this lab we will learn:

1. how signals work
2. how to use signals to communicate between processes

## 2 Signals

The operating system often needs to communicate with a running program. Often when a program is running it needs to be delivered some message. This is the scenario where one would employ signals.

The arrival of a signal is treated by a program like the arrival of an interrupt. [1] Just like on the arrival of a hardware interrupt, a program will leave whatever it was doing and jump to the interrupt handler, similarly, on the arrival of a signal the program will leave whatever it was doing and jump to a *signal handler*. The signal handler code will be executed and on return from the signal handler, the process will resume its execution from where it was interrupted. Figure 1 [1] shows a signal being handled.

Signals are given names starting with the SIG... prefix. Internally they are defined as unsigned integer values. Many unix-like OSes define some 30-ish different signals. [2]

Usually an OS delivers signals to a process as a result of OS or hardware events. Other processes or, indeed, a user can also send signals to a process via the kernel. One of the common signals a user sends to a process is `SIGINT`, i.e., a terminal interrupt signal. When a program becomes non-responsive or goes in an infinite loop we usually type `Ctrl+C` to terminate it; that's basically we telling the terminal to tell the OS to send the `SIGINT` signal to that program.

A program can tell the kernel to do one the following things on arrival of a signal:

1. Do the default action. Every signal has a default action associated with it. If the process does not specify anything special, on arrival of a signal, the operating system will perform the default action for that process which in most cases is to terminate the process.

[1] Signals are software interrupts. [2].

[2] See `https://en.wikipedia.org/wiki/Signal_(IPC)` for an exhaustive description or one of the two references given at the end.

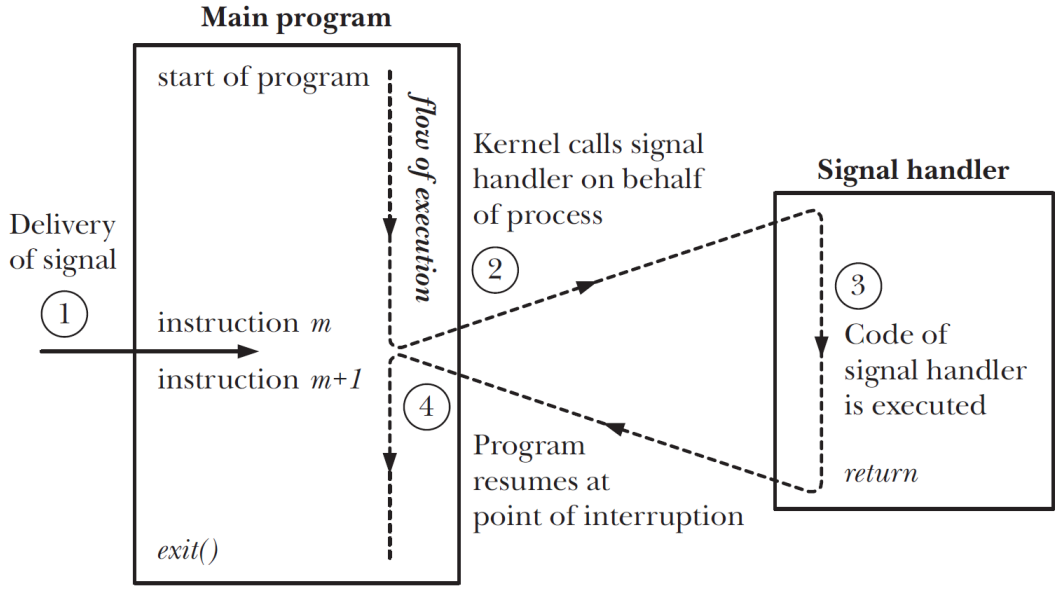


**Figure 20-1:** Signal delivery and handler execution

Figure 1: Signal delivery and handling

2. Ignore the signal. A process can tell the kernel to do nothing when a signal arrives, not even the default action, effectively ignoring the signal. `SIGKILL` and `SIGSTOP` can never be ignored.

3. Catch the signal. A process can designate one of its functions as a *signal handler* for a particular signal. When specified like this, the kernel will call the signal handler function on behalf of that process whenever that particular signal arrives. In the signal handler function, the process can do whatever it wants to do on arrival of that signal. `SIGKILL` and `SIGSTOP` cannot be caught.

Let us try handling this `SIGINT` signal. The code listed in listing 1 runs an infinite loop. Compile and run it in a terminal and then press `Ctrl+C` to terminate it.

```c
#include <stdio.h>
#include <unistd.h>  // for sleep()


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

  for (j = 0; ; j++) {
    printf("%d\n", j);
    sleep(3);
  }
  /* Loop slowly... */
}
```

Listing 1: Infinite loop (loop.c)

## 3 Ignoring a signal

Next we will configure our program to ignore this signal. The code in listing 2 calls the `signal()` [3] function to tell the kernel to ignore `SIGINT` for this process. The `signal()` function takes two arguments: the name of a signal and the action to perform on its occurrence. `SIG_IGN` flag is used to ignore a signal, i.e., do nothing on its occurrence. Compile and run the code in listing 2 and try terminating it again by pressing `Ctrl+C`.

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>  // for sleep()
#include <signal.h>


int
main(int argc, char *argv[])
{
  int j;
  if (signal(SIGINT, SIG_IGN) == SIG_ERR){
    fprintf(stderr, "Error ignoring signal\n");
    exit (1);
  }

  for (j = 0; ; j++) {
    printf("%d\n", j);
    sleep(3);
  }
  /* Loop slowly... */
}
```

Listing 2: Ignoring SIGINT (signal0.c)

Now find a way to terminate it ☺.

## 4 Catching a signal

Another behaviour we can have on the arrival of a signal is to *catch* it and then perform some action on its occurrence. For that we will have to define a function as the signal handler for that signal. The code in listing 3 defines a function called `sig_handler()` and registers it as the *signal handler* [4] for the signal `SIGINT`.

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>  // for sleep()
#include <signal.h>

//#include "tlpi_hdr.h"

static void
sig_handler(int sig)
{
  printf("Ouch!\n");
}


int
main(int argc, char *argv[])
{
  int j;
  if (signal(SIGINT, sig_handler) == SIG_ERR){
```

[3] See the man page for the `signal()` function for details.
[4] the signal handler should have the signature `void fn (int)`.

```
20      fprintf(stderr, "signal handler not registered\n");
21      exit (1);
22    }
23
24    for (j = 0; ; j++) {
25      printf("%d\n", j);
26      sleep(3);
27    }
28    /* Loop slowly... */
29  }
```

Listing 3: Catching SIGINT (signal1.c)

Compile and run the code and then try terminating it by pressing `Ctrl+C`. Note thebehaviour every time you press `Ctrl+C`.

# 5 Todo: Send signals between processes

Signals can be sent to a process either as a result of OS or hardware events. Other processes or indeed a user can also send signals to a process via the kernel.

The linux API provides the `kill()` function to let one process send a signal to another process. It takes the `PID` of the process to whom we want to send the signal and the signal we want to send.

```
int kill(pid_t pid, int signo);
```
[5]

Use the `kill()` function to send signals to another process. The receiving process should have the code to catch the signals that you will be sending.

Since the sender process will be needing the PID of the receiver process, making one ofthem the child of the other may make things easier.

Submit the code `*.c files` as well as the PDF.

# References


[1] M. Kerrisk. *The Linux Programming Interface*. No Starch Press Series. No Starch Press, 2010. ISBN: 9781593272203. URL: `https://books.google.com.pk/books?id=2SAQAQAAQBAJ`.

[2] W.R. Stevens and S.A. Rago. *Advanced Programming in the UNIX Environment: Advanc Progra UNIX Envir_p3*. Addison-Wesley Professional Computing Series. Pearson Education, 2013. ISBN: 9780321638007. URL: `https://books.google.gl/books?id=kCTMFpEcIOwC`.


[5] read the man page for a full description.