Chapter 5

IPC: Signals

Signals are an inter-process communication mechanism provided by operating systems which facilitate communication between processes. It is usually used for notifying a process regarding a certain event. So, we can say that signals are an event-notification system provided by the operating system.

5.1 Signal Delivery Using Kill

Kill is the delivery mechanism for sending a signal to a process. Unlike the name, a kill() call is used only to send a signal to a process. It does not necessarily mean that a process is going to be killed (although it can do exactly that as well). As mentioned earlier, the signal facility is just an event notification facility provided by the operating system. For e.g., a shutdown signal (SIGHUP) can be sent to all processes currently active in the system in order to notify them about a shutdown process event. Upon receipt of this signal, all processes will prepare to terminate. Once the processes terminate, the system can shutdown.

The Kill facility can be used in two ways; as a command from your command prompt, or via the kill() call from your program.

5.1.1 Kill Command

The syntax of the kill command is as such:

```
kill -s PID
```

Here, kill is the command itself, -s is an argument which specifies the type of signal to send, and PID is the integer identifier of the process to which a signal is going to be delivered. The list of signals for -s argument can be seen from the following:

```
kill -l
```

Hence, supposing that we want to terminate a process, we may enter

```
kill -9 12345
```

Where 12345 will be a process id of any active process in the system.

5.1.1.1 Exercise

Let us kill our bash shell using the TERM signal. Find out:

- The integer representation for the SIGTERM signal
- The PID of your current active bash shell using the ps command

Then, use the kill command to send over this signal.

5.1.2 Kill() Call

Usage of the Kill() system call is as such:

```
kill(int, int);
```

For this to work, we will require the *sys/types.h* & *signal.h* C libraries. Here, the 1st parameter is the integer number of signal type (again, can be checkable from *kill -l*), and the 2nd parameter is the PID of the process to which signal is to be delivered. As an example, if we want to send the Terminate signal to the current process, we will use

~~`kill(SIGTERM, getpid())`~~
kill(getpid( ), SIGTERM)

or

~~`kill(9, getpid())`~~
kill(getpid( ), 9)

5.1.2.1 Exercise

Write code which does the following:

- Parent process creates a child process using *fork()* call (using proper if/else statement blocks).
- Parent sends the terminate signal via kill() call to child and then waits for 120 seconds

While parent is waiting, the user should check the outcome of ***ps*** command.

5.1.2.2 Exercise

Modify the code in Exercise-5.1.2.1 such that signal is sent from child to parent. and then have your child wait for 120 seconds.

Again, while child is waiting, the user should check the outcome of ***ps*** command.

5.2 Signal Handling Using Signal

Signal delivery is handled by the kill command or the kill() call. The process receiving the signal can behave in a number of ways, which is defined by the signal() call.

The syntax of the call is as such:

```
signal(int, conditions)
```

The signal will require the *signal.h* C library to work. From the syntax above, signal() is the system call, the 1st parameter *int* is the integer identifier of the respective signal which is sent, and the last parameter is either of the following:

- SIG_DFL which will perform the default mechanism provided by the operating system for that particular signal
- SIG_IGN which will ignore that particular signal if it is delivered
- Any function name (for programmer-defined signal handling purposes)

As an example, we are going to send a process the SIGINT signal and count the total number of times that it is received. See the code below for this purpose:

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

int sigCounter = 0;

void sigHandler(int sigNum)
{
 printf("Signal received is %d\n", sigNum);
 ++sigCounter;
 printf("Signals received %d\n", sigCounter);
}

int main()
{
 signal(SIGINT, sigHandler);
 while(1)
 {
  printf("Hello Dears\n");
  sleep(1);
 }
 return 0;
}
```

Here, signal is the signal handler call and accepts as input the signal number, and the name of function which is going to behave as signal handler. (If we want it to behave the default way, we specify SIG_DFL, or if we want it to ignore a certain signal, we specify SIG_IGN).

When we run the program, we are instructing our program that if in case the SIGINT signal is detected, we will perform the steps provided in the *sigHandler* function. As long as there is no event, the program will keep on executing the while loop. The event can be delivered by pressing **CTRL+C**.

Try pressing CTRL+C with, and without the signal() call and you will understand the difference yourselves.