What Are Signals?

Signals are various notifications sent to a process in order to notify it of various "important" events. By their nature, they interrupt whatever the process is doing at this minute, and force it to handle them immediately. Each signal has an integer number that represents it (1, 2 and so on), as well as a symbolic name that is usually defined in the file “signal.h”. Each signal may have a signal handler, which is a function that gets called when the process receives that signal. When a signal is sent to a process, the operating system stops the execution of the process, and forces it to call the signal handler function.When that signal handler returns, the process continues execution from wherever it stopped.

Here is a list of signals that are used with process control:

- SIGKILL: Kill a process (cannot be caught or ignored).
- SIGSTOP: Stop a process (cannot be caught or ignored).
- SIGTSTP: Interactive stop (Ctrl-Z).
- SIGCONT: Continue a stopped process.
- SIGCHLD: Sent by child to its parent on exit
- SIGINT: Interrupt a process (Ctrl-C), default action is terminating the process.
- There are 2 user defined signals SIGUSR1, and SIGUSR2. These signals can be used by users for any purpose. (Default handler is to terminate a process)

P.S to get complete list of signals try kill -l on your terminal

Signals Action

There are three different things that we can tell the kernel to do when a signal occurs. We call this ‘the disposition of the signal or the action associated with a signal’.

- Ignore the signal: This works for most signals, but there are two signals that can never be ignored: SIGKILL and SIGSTOP.
- Catch the signal: We specify to the operating system a user-supplied function that should be called on delivery of the signal (Like Interrupt Service Routine in DOS).
- Use the default action: The runtime environment sets up a set of default signal handlers for each process. For example, the default signal handler for the TERM signal calls the exit() system call

Child Relations

When a process forks another one the child process inherits the parent’s signal dispositions.

Sending Signals to Processes

- Using keyboard: Ctrl-C sends SIGINT to the running process, and Ctrl-Z sends SIGTSTP.
- Using the command line: The kill command can be used to send any signal to end process.
    - Example:kill -9 <PID>
    - P.S. You can use the “ps” command to find the PID of any process
- Using system calls: The kill system call can also be used to send signals.
    - Example: kill(my_pid, SIGSTOP); //send to process with id = my_pid
    - Example: raise(SIGSTOP); //send to itself

How to use your own signal handlers

There are 3 steps that gets this job done:

1. Declare and implement the signal handler

```
void myHandler(int sig_num)
{
        // Do some stuff
        printf("Hello, I'm the new signal handler");
}
```

2. Let the program know, that you want to use this handler (myHandler) to handle a signal of type X

```
main () {
        // Some code
        signal(SIGINT, myHandler);
        // Some code
}
```

The signal() system call is used to set a signal handler for a signal (e.g. SIGINT, SIGUSR1)

3. Try it out!

```
main () {
        // Some code
        signal(SIGINT, myHandler);
        raise(SIGINT)
        // Some code
}
```

When you run the program it should execute the new signal handler which prints the hello line

Notes:

- On some UNIX/Linux systems, when a signal handler is called, the system automatically resets the signal handler for that signal to the default handler. Thus, we re-assign the signal handler immediately when entering the handler function. Otherwise, the next time this signal is received, the default handler will be executed. Even on systems that do not behave in this way, it still won't hurt, so adding this line always is a good idea. Example:

```
void myHandler(int sig_num)
{
        // Do some stuff
        printf("Hello, I'm the new signal handler");
        signal(SIGINT, myHandler);
}
```

- There are two pre-defined signal handler functions that we can use, instead of writing our own: SIG_IGN and SIG_DFL. SIG_IGN causes the process to ignore the specified signal. SIG_DFL causes the system to set the default signal handler for the given signal.

Functions Wrap-up

int kill(pid_t *pid*, int *sig*); **Description**: The **kill**() system call can be used to send any signal to any process group or process. If *pid* is positive, then signal *sig* is sent to the process with the ID specified by *pid*. If *pid* equals 0, then *sig* is sent to every process in the process group of the calling process. sig can be any of the signal stated above (e.g. SIGCHLD)
• kill() and raise(). • killpg(), getpgrp(). • alarm() and pause(). • signal().
int raise(int *sig*); **Description**: sends a signal to the calling process or thread. In a single-threaded program it is equivalent to kill(getpid(), sig). sig can be any of the above signals.
int killpg(int *pgrp*, int *sig*); **Description**: sends the signal *sig* to the process group *pgrp*. If *pgrp* is 0, **killpg**() sends the signal to the calling process's process group. Get the calling process group id using getpgrp()
unsigned int alarm(unsigned int seconds**);** **Description:** causes the system to generate a SIGALRM signal for the process after the number of realtime seconds specified by *seconds* have elapsed. If there is a previous *alarm*() request with time remaining, *alarm*() shall return a non-zero value that is the number of seconds until the previous request would have generated a SIGALRM signal. Otherwise, *alarm*() shall return 0.
int pause(void); **Description:** causes the calling process (or thread) to sleep until a signal is delivered that either terminates the process or causes the invocation of a signal-catching function
sighandler_t signal(int *signum*, sighandler_t *handler*); **Description:** sets the disposition of the signal *signum* to *handler*, which is either SIG_IGN, SIG_DFL, or the address of a programmer-defined function (a "signalhandler").

Implementing Timers Using Signals

Timers are important to allow one to check timeouts (e.g. wait for user input up to 30 seconds), check some conditions on a regular basis (e.g. check every 30 seconds that a server still active), and so on. The operating system gives us a simple way of setting up timers by using special alarm signals. They are generally limited to one timer active at a time, but that will suffice in simple cases. The alarm() system call is used to ask the system to send our process a special signal, named ALRM, after a given number of seconds. Lets see an example of how to use the ALRM signal:

```
void alarm_handler(int sig_num)
{
printf("Operation timed out. Exiting...\n\n");
exit(0);
}
/* and inside the main program... */
printf("User:"); /* prompt the user for input */
alarm(30); /* start a 30 seconds alarm */
gets(user); /* wait for user input */
alarm(0); /* remove the timer if we got the user's input */
```

Experiment Steps:

1. signal01.c contains a simple program with an infinite loop. Compile and run it in background. Practice using the kill command to send different signals to the process.

2. signal02.c illustrates an example of installing a user defined handler for SIGINT. Run the program in background and press Ctrl-C and see what is happening.

3. Another example of installing a user defined signal handler is illustrated in signal02.c. The handler catches the signal SIGCHLD (the termination of a child process).

4. signal04.c illustrates the use of raise() function (used for self signaling).

5. signal05.c shows how child process inherits the disposition of signals form its parent. Run the program and press Ctrl-C to see the effect.

6. Sending signals to a group of processes using killpg() and getpgrp() functions is illustrated in signal06.c.

7. Using timers and alarm signals is illustrated in signal07.c.