Signal Handling in Linux

Tushar B. Kute,
http://tusharkute.com
What is a Signal?

• A signal is an asynchronous event which is delivered to a process.
• Asynchronous means that the event can occur at any time may be unrelated to the execution of the process.
• Signals are raised by some error conditions, such as memory segment violations, floating point processor errors, or illegal instructions.
  – e.g. user types ctrl-C, or the modem hangs
Signal Sources

- Shell command
- Terminal driver
- Kernel
- Memory management
- Window manager
- Other user processes

Signals:
- SIGINT
- SIGHUP
- SIGKILL
- SIGWINCH
- SIGQUIT
- SIGPIPE
- SIGALRM
- SIGUSR1
POSIX predefined signals

- **SIGALRM**: Alarm timer time-out. Generated by `alarm()` API.
- **SIGABRT**: Abort process execution. Generated by `abort()` API.
- **SIGFPE**: Illegal mathematical operation.
- **SIGHUP**: Controlling terminal hang-up.
- **SIGILL**: Execution of an illegal machine instruction.
- **SIGINT**: Process interruption. Can be generated by <Delete> or <ctrl_C> keys.
- **SIGKILL**: Sure kill a process. Can be generated by
  - “kill -9 <process_id>“ command.
- **SIGPIPE**: Illegal write to a pipe.
- **SIGQUIT**: Process quit. Generated by <crtl_\> keys.
- **SIGSEGV**: Segmentation fault. generated by de-referencing a NULL pointer.
POSIX predefined signals

- **SIGTERM**: process termination. Can be generated by
  - “kill <process_id>” command.
- **SIGUSR1**: Reserved to be defined by user.
- **SIGUSR2**: Reserved to be defined by user.
- **SIGCHLD**: Sent to a parent process when its child process has terminated.
- **SIGCONT**: Resume execution of a stopped process.
- **SIGSTOP**: Stop a process execution.
- **SIGTTIN**: Stop a background process when it tries to read from its controlling terminal.
- **SIGTSTP**: Stop a process execution by the control_Z keys.
- **SIGTTOUS**: Stop a background process when it tries to write to its controlling terminal.
Actions on signals

- Process that receives a signal can take one of three action:
  - Perform the system-specified default for the signal
    - notify the parent process that it is terminating;
    - generate a core file; (a file containing the current memory image of the process)
    - terminate.
  - Ignore the signal
    - A process can do ignoring with all signal but two special signals: SIGSTOP and SIGKILL.
  - Catch the Signal
    - When a process catches a signal, except SIGSTOP and SIGKILL, it invokes a special signal handing routine.
• User types Ctrl-c
  – Event gains attention of OS
  – OS stops the application process immediately, sending it a 2/SIGINT signal
  – Signal handler for 2/SIGINT signal executes to completion
  – Default signal handler for 2/SIGINT signal exits process

• Process makes illegal memory reference
  – Event gains attention of OS
  – OS stops application process immediately, sending it a 11/SIGSEGV signal
  – Signal handler for 11/SIGSEGV signal executes to completion
  – Default signal handler for 11/SIGSEGV signal prints “segmentation fault” and exits process
Send signals via commands

- **kill Command**
  - `kill -signal pid`
    - Send a signal of type signal to the process with id pid
    - Can specify either signal type name (-SIGINT) or number (-2)
    - *No signal type name or number specified => sends 15/SIGTERM signal*
  - Default 15/SIGTERM handler exits process
    - Better command name would be sendsig
- **Examples**
  - `kill -2 1234`
  - `kill -SIGINT 1234`
    - Same as pressing Ctrl-c if process 1234 is running in foreground
```c
#include<stdio.h>
int main()
{
    while(1)
        printf("Hello World...
");
    return 0;
}
```
Check the output
Check the output

- Go to new terminal and check the process list (`ps -aux`)

```
+-----------------+-----------------+------------------+
|   User   |   PID   |   Command          |
+-----------------+-----------------+------------------+
| sitrc           | 5961  | /usr/lib/i386-linux-gnu/unitl     |
| sitrc           | 5976  | /usr/lib/gvfs/gvfsd-http -s  |
| root            | 7270  | /bin/sh               |
| root            | 7271  | /bin/sh               |
| root            | 7272  | /bin/sh               |
| root            | 7273  | /bin/sh               |
| root            | 7274  | /bin/sh               |
| sitrc           | 7320  | /usr/lib/i386-linux-gnu/unitl     |
| sitrc           | 7321  | /usr/lib/i386-linux-gnu/unitl     |
| root            | 7275  | /usr/lib/i386-linux-gnu/unitl     |
| root            | 7276  | /usr/lib/i386-linux-gnu/unitl     |
| root            | 7277  | /usr/lib/i386-linux-gnu/unitl     |
| root            | 7278  | /usr/lib/i386-linux-gnu/unitl     |
| root            | 7279  | /usr/lib/i386-linux-gnu/unitl     |
| root            | 7280  | /usr/lib/i386-linux-gnu/unitl     |
| sitrc           | 8493  | /dev/sda2             |
| sitrc           | 8558  | /dev/sda2             |
```

pid
Kill the process

- `kill 8493`
Killing process by different signals

- `kill -SIGSEGV 8493`

```
Segmentation fault (core dumped)
```

SIGSEGV signal received
Signal Concepts

• Signals are defined in <signal.h>
• **man 7 signal** for complete list of signals and their numeric values.
• **kill –l** for full list of signals on a system.
• 64 signals. The first 32 are traditional signals, the rest are for real time applications
Programs can handle signals using the signal library function.

```c
void (*signal(int signo, void (*func)(int)))(int);
```

- `signo` is the signal number to handle
- `func` defines how to handle the signal
  - `SIG_IGN`
  - `SIG_DFL`
  - Function pointer of a custom handler
- Returns previous disposition if ok, or `SIG_ERR` on error
Example:

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

void ohh(int sig)
{
    printf("Ohh! - I got signal %d\n", sig);
    (void) signal(SIGINT, SIG_DFL);
}

int main()
{
    (void) signal(SIGINT, ohh);
    while(1)
    {
        printf("Hello World!\n");
        sleep(1);
    }
    return 0;
}
```
```
tushar@tushar-laptop ~ $ gcc sig1.c
tushar@tushar-laptop ~ $ ./a.out
Hello World!
Hello World!
Hello World!
^C
```

```
OHh! - I got signal 2
Hello World!
Hello World!
^C
```

```
tushar@tushar-laptop ~ $
```

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

void error(int sig)
{
    printf("Ohh! its a floating point error...\n");
    (void) signal(SIGFPE, SIG_DFL);
}

int main()
{
    (void) signal(SIGFPE, error);
    int a = 12, b = 0, result;
    result = a / b;
    printf("Result is : %d\n", result);
    return 0;
}
```
tushar@tushar-laptop ~ $ gcc sig2.c

Ohh! its a floating point error...
Floating point exception

tushar@tushar-laptop ~ $
The `sigaction` function sets the action associated with the signal `sig`. If `oact` is not null, `sigaction` writes the previous signal action to the location it refers to. If `act` is null, this is all `sigaction` does. If `act` isn’t null, the action for the specified signal is set.
As with signal, sigaction returns 0 if successful and -1 if not. The error variable `errno` will be set to EINVAL if the specified signal is invalid or if an attempt is made to catch or ignore a signal that can’t be caught or ignored.

Within the sigaction structure pointed to by the argument `act`, `sa_handler` is a pointer to a function called when signal `sig` is received. This is much like the function `func` you saw earlier passed to signal.

You can use the special values `SIG_IGN` and `SIG_DFL` in the `sa_handler` field to indicate that the signal is to be ignored or the action is to be restored to its default, respectively.
void ohh(int sig)
{
    printf("Ohh! - I got signal %d\n", sig);
}

int main()
{
    struct sigaction act;
    act.sa_handler = ohh;
    sigemptyset(&act.sa_mask);
    act.sa_flags = 0;
    sigaction(SIGINT, &act, 0);
    while(1)
    {
        printf("Hello World!\n");
        sleep(1);
    }
}
tushar@tushar-laptop ~ $ gcc sig3.c
Hello World!
Hello World!
Hello World!
^C0hh! - I got signal 2
Hello World!
^C0hh! - I got signal 2
Hello World!
Hello World!
Hello World!
^C0hh! - I got signal 2
Hello World!
Hello World!

Output
Implement the C program to demonstrate the use of SIGCHLD signal. A parent process creates multiple child processes (minimum three child processes). Parent process should be sleeping until it creates the number of child processes. Child processes send SIGCHLD signal to parent process to interrupt from the sleep and force the parent to call wait for the Collection of status of terminated child processes.
**Program**

```c
void handler(int sig)
{
    pid_t pid;
    pid = wait(NULL);
    printf("\tChild %d exited.\n", pid);
    signal(SIGCHLD, handler);
}

int main()
{
    int i;
    signal(SIGCHLD, handler);
    for(i=0;i<3;i++)
        switch(fork())
        {
        case 0:
            printf("\tChild created %d\n", getpid());
            exit(0);
        }
    sleep(2);
    return 0;
}
```
tushar@tushar-laptop ~ $ ./a.out
Parent created 5928
   Child created 5929
Parent created 5928
   Child created 5930
       Child 5929 exited.
Parent created 5928
   Child 5930 exited.
   Child created 5932
       Child 5932 exited.
This presentation is created using LibreOffice Impress 4.2.7.2, can be used freely as per GNU General Public License