

CS232L Operating Systems Lab

Lab 14: Signals in Depth

CS Program

Habib University

1. Introduction:

This lab gives you a lot more details about various different signals. For this lab you are required to:

1. Implement Code of each example and task shared.
2. Make a PDF file explaining the different uses of all the signals covered in this lab.

2. Generating Signals

Every signal has a symbolic name starting with `SIG`. The signal names are defined in `signal.h`, which must be included by any C program that uses signals. The names of the signals represent small integers greater than 0. [Table 1](#) describes the required POSIX signals and lists their default actions. Two signals, `SIGUSR1` and `SIGUSR2`, are available for users and do not have a preassigned use. Some signals such as `SIGFPE` or `SIGSEGV` are generated when certain errors occur; other signals are generated by specific calls such as `alarm`.

Table 1. The POSIX required signals.

| signal | description | default action |
|----------------------|--|--------------------------|
| <code>SIGABRT</code> | process abort | implementation dependent |
| <code>SIGALRM</code> | alarm clock | abnormal termination |
| <code>SIGBUS</code> | access undefined part of memory object | implementation dependent |
| <code>SIGCHLD</code> | child terminated, stopped or continued | ignore |
| <code>SIGCONT</code> | execution continued if stopped | continue |
| <code>SIGFPE</code> | error in arithmetic operation as in division by zero | implementation dependent |
| <code>SIGHUP</code> | hang-up (death) on controlling terminal (process) | abnormal termination |

| | | |
|----------------------|---|--------------------------|
| <code>SIGILL</code> | invalid hardware instruction | implementation dependent |
| <code>SIGINT</code> | interactive attention signal (usually Ctrl-C) | abnormal termination |
| <code>SIGKILL</code> | terminated (cannot be caught or ignored) | abnormal termination |
| <code>SIGPIPE</code> | write on a pipe with no readers | abnormal termination |
| <code>SIGQUIT</code> | interactive termination: core dump (usually Ctrl-) | implementation dependent |
| <code>SIGSEGV</code> | invalid memory reference | implementation dependent |
| <code>SIGSTOP</code> | execution stopped (cannot be caught or ignored) | stop |
| <code>SIGTERM</code> | termination | abnormal termination |
| <code>SIGTSTP</code> | terminal stop | stop |
| <code>SIGTTIN</code> | background process attempting to read | stop |
| <code>SIGTTOU</code> | background process attempting to write | stop |

| | | |
|----------------------|---|----------------------|
| <code>SIGURG</code> | high bandwidth data available at a socket | ignore |
| <code>SIGUSR1</code> | user-defined signal 1 | abnormal termination |
| <code>SIGUSR2</code> | user-defined signal 2 | abnormal termination |

Generate signals from the shell with the `kill` command. The name `kill` derives from the fact that, historically, many signals have the default action of terminating the process. The `signal_name` parameter is a symbolic name for the signal formed by omitting the leading `SIG` from the corresponding symbolic signal name.

SYNOPSIS

```
kill -s signal_name pid...
kill -l [exit_status]
```

Example 1

The following command is the traditional way to send signal number 9 (`SIGKILL`) to process 3423.

```
kill -9 3423
```

Example 2

The following command sends the `SIGUSR1` signal to process 3423.

```
kill -s USR1 3423
```

Kill -1 command:

The `kill -1` command gives a list of the available symbolic signal names. A system running Sun Solaris produced the following sample output.

```
% kill -1
HUP INT QUIT ILL TRAP ABRT EMT FPE
KILL BUS SEGV SYS PIPE ALRM TERM USR1
USR2 CLD PWR WINCH URG POLL STOP TSTP
CONT TTIN TTOU VTALRM PROF XCPU XFSZ WAITING
LWP FREEZE THAW CANCEL LOST XRES RTMIN RTMIN+1
RTMIN+2 RTMIN+3 RTMAX-3 RTMAX-2 RTMAX-1 RTMAX
```

Task:

Call the `kill` function in a program to send a signal to a process. The `kill` function takes a process ID and a signal number as parameters. If the `pid` parameter is greater than zero, `kill` sends the signal to the process with that ID. If `pid` is 0, `kill` sends the signal to members of the caller's process group. If the `pid` parameter is -1, `kill` sends the signal to all processes for which it has permission to send. If the `pid` parameter is another negative value, `kill` sends the signal to the process group with group ID equal to `|pid|`.

SYNOPSIS

```
#include <signal.h>

int kill(pid_t pid, int sig);
```

If successful, `kill` returns 0. If unsuccessful, `kill` returns -1 and sets `errno`. The following table lists the mandatory errors for `kill`.

| errno | cause |
|--------|---|
| EINVAL | <code>sig</code> is an invalid or unsupported signal |
| EPERM | caller does not have the appropriate privileges |
| ESRCH | no process or process group corresponds to <code>pid</code> |

A user may send a signal only to processes that he or she owns. For most signals, `kill` determines permissions by comparing the user IDs of caller and target. `SIGCONT` is an exception. For `SIGCONT`, user IDs are not checked if `kill` is sent to a process that is in the same session.

Example 4

The following code segment sends `SIGUSR1` to process 3423.

```
if (kill(3423, SIGUSR1) == -1)
    perror("Failed to send the SIGUSR1 signal");
```

Normally, programs do not hardcode specific process IDs such as 3423 in the `kill` function call. The usual way to find relevant process IDs is with `getpid`, `getppid`, `getgid` or by saving the return value from `fork`.

Example 5

This scenario sounds grim, but a child process can kill its parent by executing the following code segment.

```
if (kill(getppid(), SIGTERM) == -1)
    perror ("Failed to kill parent");
```

A process can send a signal to itself with the `raise` function. The `raise` function takes just one parameter, a signal number.

SYNOPSIS

```
#include <signal.h>

int raise(int sig);
```

If successful, `raise` returns 0. If unsuccessful, `raise` returns a nonzero error value and sets `errno`. The `raise` function sets `errno` to `EINVAL` if `sig` is invalid.

Example 6

The following statement causes a process to send the `SIGUSR1` signal to itself.

```
if (raise(SIGUSR1) != 0)
    perror("Failed to raise SIGUSR1");
```

A key press causes a hardware interrupt that is handled by the device driver for the

keyboard. This device driver and its associated modules may perform buffering and editing of the keyboard input. Two special characters, the `INTR` and `QUIT` characters, cause the device driver to send a signal to the foreground process group. A user can send the `SIGINT` signal to the foreground process group by entering the `INTR` character. This user-settable character is often `Ctrl-C`. The user-settable `QUIT` character sends the `SIGQUIT` signal.

Some More Signals

The `QUIT` character (`Ctrl-\`) generates `SIGQUIT`. The `SUSP` character (`Ctrl-Z`) generates `SIGSTOP`, and the `DSUSP` character (`Ctrl-Y`) generates `SIGCONT`.

Example 7

`simplealarm.c`

Since the default action for `SIGALRM` is to terminate the process, the following program runs for approximately ten seconds of wall-clock time.

```
#include <unistd.h>

int main(void) {
    alarm(10);
    for ( ; ; ) ;
}
```

3. Blocking and Unblocking a Signal

A process can temporarily prevent a signal from being delivered by blocking it. Blocked signals do not affect the behavior of the process until they are delivered. The process *signal mask* gives the set of signals that are currently blocked. The signal mask is of type `sigset_t`.

Blocking a signal is different from ignoring a signal. When a process blocks a signal, the operating system does not deliver the signal until the process unblocks the signal. A process blocks a signal by modifying its signal mask with `sigprocmask`. When a process ignores a signal, the signal is delivered and the process handles it by throwing it away. The process sets a signal to be ignored by calling `sigaction` with a handler of `SIG_IGN`.

Signal sets are manipulated by the five functions listed in the following synopsis box. The first parameter for each function is a pointer to a `sigset_t`. The `sigaddset` adds `signo` to the signal set, and the `sigdelset` removes `signo` from the signal set. The `sigemptyset` function initializes a `sigset_t` to contain no signals; `sigfillset` initializes a `sigset_t` to contain all signals. Initialize a signal set by calling either `sigemptyset` or `sigfillset` before using it. The `sigismember` reports whether `signo` is in a `sigset_t`.

SYNOPSIS

```
#include <signal.h>

int sigaddset(sigset_t *set, int signo);
int sigdelset(sigset_t *set, int signo);
int sigemptyset(sigset_t *set);
int sigfillset(sigset_t *set);
int sigismember(const sigset_t *set, int signo);
```

The `sigismember` function returns 1 if `signo` is in `*set` and 0 if `signo` is not in `*set`. If successful, the other functions return 0. If unsuccessful, these other functions return -1 and set `errno`.

Example 8

The following code segment initializes signal set `twosigs` to contain exactly the two signals `SIGINT` and `SIGQUIT`.

```
if ((sigemptyset(&twosigs) == -1) ||
    (sigaddset(&twosigs, SIGINT) == -1) ||
    (sigaddset(&twosigs, SIGQUIT) == -1))
    perror("Failed to set up signal mask");
```

A process can examine or modify its process signal mask with the `sigprocmask` function. The `how` parameter is an integer specifying the manner in which the signal mask is to be modified.

The `set` parameter is a pointer to a signal set to be used in the modification. If `set` is `NULL`, no modification is made. If `oset` is not `NULL`, the `sigprocmask` returns in `*oset` the signal set before the modification.

SYNOPSIS

```
#include <signal.h>

int sigprocmask(int how, const sigset_t *restrict set,
                sigset_t *restrict oset);
```

If successful, `sigprocmask` returns 0. If unsuccessful, `sigprocmask` returns -1 and sets `errno`. The `sigprocmask` function sets `errno` to `EINVAL` if `how` is invalid. The `sigprocmask` function should only be used by a process with a single thread. When multiple threads exist, the `pthread_sigmask` function should be used.

The `how` parameter, which specifies the manner in which the signal mask is to be modified, can take on one of the following three values.

`SIG_BLOCK`: add a collection of signals to those currently blocked

`SIG_UNBLOCK`: delete a collection of signals from those currently blocked

`SIG_SETMASK`: set the collection of signals being blocked to the specified set

Keep in mind that some signals, such as `SIGSTOP` and `SIGKILL`, cannot be blocked. If an attempt is made to block these signals, the system ignores the request without reporting an error.

Example 9

The following code segment adds `SIGINT` to the set of signals that the process has blocked.

```
sigset_t newsigset;

if ((sigemptyset(&newsigset) == -1) ||
    (sigaddset(&newsigset, SIGINT) == -1))
    perror("Failed to initialize the signal set");
else if (sigprocmask(SIG_BLOCK, &newsigset, NULL) == -1)
    perror("Failed to block SIGINT");
```

If `SIGINT` is already blocked, the call to `sigprocmask` has no effect.

Task:

Write a program that displays a message, blocks the `SIGINT` signal while doing some work, unblocks the signal, and does more work. The program repeats this sequence continually in a loop.

If a user enters Ctrl-C while `SIGINT` is blocked, program finishes the calculation and prints a message before terminating. If a user types Ctrl-C while `SIGINT` is unblocked, the program terminates immediately.

The function `makepair` of code given below takes two pathnames as parameters and creates two named pipes with these names. If successful, `makepair` returns 0. If unsuccessful, `makepair` returns -1 and sets `errno`. The function blocks all signals during the creation of the two pipes to be sure that it can deallocate both pipes if there is an error. The function restores the original signal mask before the return. The `if` statement relies on the conditional left-to-right evaluation of `&&` and `||`.

Answer this question in PDF:

Is it possible that after a call to `makepair`, `pipe1` exists but `pipe2` does not?

Code `makepair.c`

A function that blocks signals while creating two pipes.


```

#include <errno.h>
#include <signal.h>
#include <unistd.h>
#include <sys/stat.h>
#define R_MODE (S_IRUSR | S_IRGRP | S_IROTH)
#define W_MODE (S_IWUSR | S_IWGRP | S_IWOTH)
#define RW_MODE (R_MODE | W_MODE)

int makepair(char *pipe1, char *pipe2) {
    sigset_t blockmask;
    sigset_t oldmask;
    int returncode = 0;

    if (sigfillset(&blockmask) == -1)
        return -1;
    if (sigprocmask(SIG_SETMASK, &blockmask, &oldmask) == -1)
        return -1;
    if (((mkfifo(pipe1, RW_MODE) == -1) && (errno != EEXIST)) ||
        ((mkfifo(pipe2, RW_MODE) == -1) && (errno != EEXIST))) {
        returncode = errno;
        unlink(pipe1);
        unlink(pipe2);
    }
    if ((sigprocmask(SIG_SETMASK, &oldmask, NULL) == -1) && !returncode)
        returncode = errno;
    if (returncode) {
        errno = returncode;
        return -1;
    }
    return 0;
}

```

Answer this question in PDF:

Does a `makepair` return value of 0 guarantee that FIFOs corresponding to `pipe1` and `pipe2` are available on return?

Task:

Write a program in which the parent blocks all signals before forking a child process to execute an `ls` command.

Code `password.c`

A function that retrieves a user password.

```

#include <errno.h>
#include <fcntl.h>

```



```

#include <signal.h>
#include <stdio.h>
#include <string.h>
#include <termios.h>
#include <unistd.h>
#include "restart.h"

int setecho(int fd, int onflag);

int password(const char *prompt, char *passbuf, int passmax) {
    int fd;
    int firsterrno = 0;
    sigset_t signew, sigold;
    char termbuf[L_ctermid];

    if (ctermid(termbuf) == NULL) {
        /* find the terminal name */  errno = ENODEV;
        return -1;
    }
    if ((fd = open(termbuf, O_RDONLY)) == -1)
        /* open descriptor to terminal */  return -1;
    if ((sigemptyset(&signew) == -1) || /* block SIGINT, SIGQUIT
                                         and SIGTSTP */
        (sigaddset(&signew, SIGINT) == -1) ||
        (sigaddset(&signew, SIGQUIT) == -1) ||
        (sigaddset(&signew, SIGTSTP) == -1) ||
        (sigprocmask(SIG_BLOCK, &signew, &sigold) == -1) ||
        (setecho(fd, 0) == -1)) {
        /* set terminal echo off */  firsterrno = errno;
        sigprocmask(SIG_SETMASK, &sigold, NULL);
        r_close(fd);
        errno = firsterrno;
        return -1;
    }
    if ((r_write(STDOUT_FILENO, (char *)prompt, strlen(prompt)) == -1) ||
        (getline(fd, passbuf, passmax) == -1))
        /* read password */  firsterrno = errno;
    else
        passbuf[strlen(passbuf) - 1] = 0; /* remove newline */
    if ((setecho(fd, 1) == -1) && !firsterrno) /* turn echo back on */
        firsterrno = errno;
    if ((sigprocmask(SIG_SETMASK, &sigold, NULL) == -1) && !firsterrno)
        firsterrno = errno;
    if ((r_close(fd) == -1) && !firsterrno) /* close descriptor to terminal */
        firsterrno = errno;
    return firsterrno ? errno = firsterrno, -1: 0;
}

```

The `password` function blocks `SIGINT`, `SIGQUIT` and `SIGTSTP` while terminal echo is set off, preventing the terminal from being placed in an unusable state if one of these signals is delivered to the process while this function is executing.

4. Catching and Ignoring Signals—`sigaction`

The `sigaction` function allows the caller to examine or specify the action associated with a specific signal. The `sig` parameter of `sigaction` specifies the signal number for the action. The `act` parameter is a pointer to a `struct sigaction` structure that specifies the action to be taken. The `oact` parameter is a pointer to a `struct sigaction` structure that receives the previous action associated with the signal. If `act` is `NULL`, the call to `sigaction` does not change the action associated with the signal. If `oact` is `NULL`, the call to `sigaction` does not return the previous action associated with the signal.

SYNOPSIS

```
#include <signal.h>

int sigaction(int sig, const struct sigaction *restrict act,
              struct sigaction *restrict oact);
```

If successful, `sigaction` returns 0. If unsuccessful, `sigaction` returns -1 and sets `errno`. The following table lists the mandatory errors for `sigaction`.

| err no | cause |
|-----------------|--|
| EI NV AL | <code>sig</code> is an invalid signal number, or attempt to catch a signal that cannot be caught, or attempt to ignore a signal that cannot be ignored |
| ENO TSU P | <code>SA_SIGINFO</code> bit of the <code>sa_flags</code> is set and the implementation does not support POSIX:RTS or POSIX:XSI |

The `struct sigaction` structure must have at least the following members.

```
struct sigaction {
    void (*sa_handler)(int); /* SIG_DFL, SIG_IGN or pointer to function */
    sigset_t sa_mask; /* additional signals to be blocked during execution
                       of handler */
    int sa_flags; /* special flags and options */
    void(*sa_sigaction) (int, siginfo_t *, void *); /* realtime handler */
};
```

The storage for `sa_handler` and `sa_sigaction` may overlap, and an application should use only one of these members to specify the action. If the `SA_SIGINFO` flag of the `sa_flags` field is cleared, the `sa_handler` specifies the action to be taken for the specified signal.

Example 10

The following code segment sets the signal handler for `SIGINT` to `mysighand`.

```
struct sigaction newact;

newact.sa_handler = mysighand; /* set the new handler */
newact.sa_flags = 0; /* no special options */
if ((sigemptyset(&newact.sa_mask) == -1) ||
    (sigaction(SIGINT, &newact, NULL) == -1))
    perror("Failed to install SIGINT signal handler");
```

A signal handler is an ordinary function that returns `void` and has one integer parameter. When the operating system delivers the signal, it sets this parameter to the number of the signal that was delivered. Most signal handlers ignore this value, but it is possible to have a single signal handler for many signals. The usefulness of signal handlers is limited by the inability to pass values to them.

Two special values of the `sa_handler` member of `struct sigaction` are `SIG_DFL` and `SIG_IGN`. The `SIG_DFL` value specifies that `sigaction` should restore the default action for the signal. The `SIG_IGN` value specifies that the process should handle the signal by ignoring it (throwing it away).

Example 11

The following code segment causes the process to ignore `SIGINT` if the default action is in effect for this signal.

```
struct sigaction act;

if (sigaction(SIGINT, NULL, &act) == -1) /* Find current SIGINT handler */
    perror("Failed to get old handler for SIGINT");
else if (act.sa_handler == SIG_DFL) { /* if SIGINT handler is default */
    act.sa_handler = SIG_IGN; /* set new SIGINT handler to ignore */
if (sigaction(SIGINT, &act, NULL) == -1)
    perror("Failed to ignore SIGINT");
}
```

Example 12

The following code segment sets up a signal handler that catches the `SIGINT` signal generated by Ctrl-C.

```
void catchctrlc(int signo) {
    char handmsg[] = "I found Ctrl-C\n";
    int msglen = sizeof(handmsg);

    write(STDERR_FILENO, handmsg, msglen);
}
```



```

...
struct sigaction act;
act.sa_handler = catchctrlc;
act.sa_flags = 0;
if ((sigemptyset(&act.sa_mask) == -1) ||
    (sigaction(SIGINT, &act, NULL) == -1))
    perror("Failed to set SIGINT to handle Ctrl-C");

```

Answer this question in PDF:

Why didn't [Example 12](#) use `fprintf` or `strlen` in the signal handler? **Example 14**

The following code segment sets the action of `SIGINT` to the default.

```

struct sigaction newact;

newact.sa_handler = SIG_DFL; /* new handler set to default */

newact.sa_flags = 0; /* no special options */

if ((sigemptyset(&newact.sa_mask) == -1) ||
    (sigaction(SIGINT, &newact, NULL) == -1))
    perror("Failed to set SIGINT to the default action");

```

Example 13 `testignored.c`

The following function takes a signal number parameter and returns 1 if that signal is ignored and 0 otherwise.

```

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

int testignored(int signo) {
    struct sigaction act;
    if ((sigaction(signo, NULL, &act) == -1) ||
        (act.sa_handler != SIG_IGN))
        return 0;
    return 1;
}

```