**Interprocess Communication**

<https://users.cs.cf.ac.uk/Dave.Marshall/C/node24.html>

**IPC: Interrupts and Signals: <signal.h>**

* The signal is discarded after being received
* The process is terminated after the signal is received
* A core file is written, then the process is terminated
* Stop the process after the signal is received

Each signal defined by the system falls into one of five classes:

* Hardware conditions
* Software conditions
* Input/output notification
* Process control
* Resource control

Macros are defined in <signal.h> header file for common signals.

These include:

|  |  |
| --- | --- |
| SIGHUP 1 /\* hangup \*/ | SIGINT 2 /\* interrupt \*/ |
| SIGQUIT 3 /\* quit \*/ | SIGILL 4 /\* illegal instruction \*/ |
| SIGABRT 6 /\* used by abort \*/ | SIGKILL 9 /\* hard kill \*/ |
| SIGALRM 14 /\* alarm clock \*/ |  |
| SIGCONT 19 /\* continue a stopped process \*/ |  |
| SIGCHLD 20 /\* to parent on child stop or exit \*/ |  |

***Signals*** can be numbered from 0 to 31.

There are two common functions used to send signals

int kill(int pid, int signal) - a system call that send a signal to a process, pid. If pid is greater than zero, the signal is sent to the process whose process ID is equal to pid. If pid is 0, the signal is sent to all processes, except system processes.

kill() returns 0 for a successful call, -1 otherwise and sets errno accordingly.

int raise(int sig) sends the signal sig to the executing program. raise() actually uses kill() to send the signal to the executing program:

kill(getpid(), sig);

Basic rule: **only processes that have the same user can send/receive messages**.

# Signal Handling -- signal()

An application program can specify a function called a signal handler to be invoked when a specific signal is received. When a signal handler is invoked on receipt of a signal, it is said to catch the signal. A process can deal with a signal in one of the following ways:

* The process can let the default action happen
* The process can block the signal (some signals cannot be ignored)
* the process can catch the signal with a handler.

Signal handlers usually execute on the current stack of the process. This lets the signal handler return to the point that execution was interrupted in the process. This can be changed on a per-signal basis so that a signal handler executes on a special stack. If a process must resume in a different context than the interrupted one, it must restore the previous context itself

Receiving signals is straightforward with the function:

int (\*signal(int sig, void (\*func)()))() -- that is to say the function signal() will call the func functions if the process receives a signal sig. Signal returns a pointer to function func if successful or it returns an error to errno and -1 otherwise.

func() can have three values:

**SIG\_DFL**

-- a pointer to a system default function SID\_DFL(), which will terminate the process upon receipt of sig.

**SIG\_IGN**

-- a pointer to system ignore function SIG\_IGN() which will disregard the sig action (UNLESS it is SIGKILL).

**A function address**

-- a user specified function.

SIG\_DFL and SIG\_IGN are defined in signal.h (standard library) header file.

Thus to ignore a ctrl-c command from the command line. we could do:

   signal(SIGINT, SIG\_IGN);

TO reset system so that SIGINT causes a termination at any place in our program, we would do:

   signal(SIGINT, SIG\_DFL);

**Other signal functions**

There are a few other functions defined in signal.h:

int sighold(int sig) -- adds sig to the calling process's signal mask

int sigrelse(int sig) -- removes sig from the calling process's signal mask

int sigignore(int sig) -- sets the disposition of sig to SIG\_IGN

int sigpause(int sig) -- removes sig from the calling process's signal mask and suspends the calling process until a signal is received

Practice Problems:

So lets write a program to trap a ctrl-c but not quit on this signal. We have a function sigproc() that is executed when we trap a ctrl-c. We will also set another function to quit the program if it traps the SIGQUIT signal so we can terminate our program:

#include <stdio.h>

#include <signal.h>

void sigproc(void);

void quitproc(void);

main()

{ signal(SIGINT, sigproc);

signal(SIGQUIT, quitproc);

printf("ctrl-c disabled use ctrl-\\ to quit\n");

for(;;); /\* infinite loop \*/}

void sigproc()

{ signal(SIGINT, sigproc); /\* \*/

/\* NOTE some versions of UNIX will reset signal to default

after each call. So for portability reset signal each time \*/

printf("you have pressed ctrl-c \n");

}

void quitproc()

{ printf("ctrl-\\ pressed to quit\n");

exit(0); /\* normal exit status \*/

}

**sig\_talk.c -- complete example program**

Let us now write a program that communicates between child and parent processes using kill() and signal().

fork() creates the child process from the parent. The pid can be checked to decide whether it is the child (== 0) or the parent (pid = child process id).

The parent can then send messages to child using the pid and kill().

The child picks up these signals with signal() and calls appropriate functions.

An example of communicating process using signals is sig\_talk.c:

/\* sig\_talk.c --- Example of how 2 processes can talk \*/

/\* to each other using kill() and signal() \*/

/\* We will fork() 2 process and let the parent send a few \*/

/\* signals to it`s child \*/

/\* cc sig\_talk.c -o sig\_talk \*/

#include <stdio.h>

#include <signal.h>

void sighup(); /\* routines child will call upon sigtrap \*/

void sigint();

void sigquit();

main()

{ int pid;

/\* get child process \*/

if ((pid = fork()) < 0) {

perror("fork");

exit(1);

}

if (pid == 0)

{ /\* child \*/

signal(SIGHUP,sighup); /\* set function calls \*/

signal(SIGINT,sigint);

signal(SIGQUIT, sigquit);

for(;;); /\* loop for ever \*/

}

else /\* parent \*/

{ /\* pid hold id of child \*/

printf("\nPARENT: sending SIGHUP\n\n");

kill(pid,SIGHUP);

sleep(3); /\* pause for 3 secs \*/

printf("\nPARENT: sending SIGINT\n\n");

kill(pid,SIGINT);

sleep(3); /\* pause for 3 secs \*/

printf("\nPARENT: sending SIGQUIT\n\n");

kill(pid,SIGQUIT);

sleep(3);

}

}

void sighup()

{ signal(SIGHUP,sighup); /\* reset signal \*/

printf("CHILD: I have received a SIGHUP\n");

}

void sigint()

{ signal(SIGINT,sigint); /\* reset signal \*/

printf("CHILD: I have received a SIGINT\n");

}

void sigquit()

{ printf("My DADDY has Killed me!!!\n");

exit(0);

}

**Exercise of the Week:**

1. **Write a chat application between two processes using signals and shared memory. User1 and User2 have created as two processes and shared memory is used to store the process id(s) of two processes. In this, handler function is used to print the message received from another process and vice versa.**

Reference:

<https://www.geeksforgeeks.org/chat-application-between-two-processes-using-signals-and-shared-memory>