Experiment Number: 08

TITLE: Inter-process Communication (IPC) in Linux using Signals.

OBJECTIVE:

1. To study use of signals in Linux.
2. To study inter-process communication using signals in Linux.

THEORY:

Signals

The common communication channel between user space program and kernel is given by the system calls. But there is a different channel, that of the signals, used both between user processes and from kernel to user process.

Sending signals

A program can signal a different program using the kill() system call with prototype

int kill(pid_t pid, int sig);

This will send the signal with number sig to the process with process ID pid. Signal numbers are small positive integers.

A user can send a signal from the command line using the kill command. Common uses are kill -9 N to kill the process with pid N, or kill -1 N to force process N (maybe init or inetd) to reread its configuration file.

Certain user actions will make the kernel send a signal to a process or group of processes: typing the interrupt character (probably Ctrl-C) causes SIGINT to be sent, typing the quit character (probably Ctrl-\) sends SIGQUIT, hanging up the phone (modem) sends SIGHUP, typing the stop character (probably Ctrl-Z) sends SIGSTOP.

Certain program actions will make the kernel send a signal to that process: for an illegal instruction one gets SIGILL, for accessing non-existing memory one gets SIGSEGV, for writing to a pipe while nobody is listening anymore on the other side one gets SIGPIPE, for reading from the terminal while in the background one gets SIGTTIN, etc.

More interesting communication from the kernel is also possible. One can ask the kernel to be notified when something happens on a given file descriptor.
A whole group of signals is reserved for real-time use.

Receiving signals

When a process receives a signal, a default action happens, unless the process has arranged to handle the signal. For the list of signals and the corresponding default actions. For example, by default SIGHUP, SIGINT, SIGKILL will kill the process; SIGQUIT will kill the process and force a core dump; SIGSTOP, SIGTTIN will stop the process; SIGCONT will continue a stopped process; SIGCHLD will be ignored.

Traditionally, one sets up a handler for the signal using the signal system call with prototype

```
typedef void (*sighandler_t)(int);
sighandler_t signal(int sig, sighandler_t handler);
```

This sets up the routine handler() as handler for signals with number sig. The return value is (the address of) the old handler. The special values SIG_DFL and SIG_IGN denote the default action and ignoring, respectively.

When a signal arrives, the process is interrupted, the current registers are saved, and the signal handler is invoked. When the signal handler returns, the interrupted activity is continued.

It is difficult to do interesting things in a signal handler, because the process can be interrupted in an arbitrary place, data structures can be in arbitrary state, etc. The three most common things to do in a signal handler are (i) set a flag variable and return immediately, and (ii) (messy) throw away all the program was doing, and restart at some convenient point, perhaps the main command loop or so, and (iii) clean up and exit.

Setting up a handler for a signal is called "catching the signal". The signals SIGKILL and SIGSTOP cannot be caught or blocked or ignored.

Semantics

The traditional semantics was: reset signal behavior to SIG_DFL upon invocation of the signal handler. Possibly this was done to avoid recursive invocations. The signal handler would do its job and at the end call signal() to establish itself again as handler.

This is really unfortunate. When two signals arrive shortly after each other, the second one will be lost if it arrives before the signal handler is called - there is no counter. And if it arrives after the signal handler is called, the default action will happen - this may very well kill the process. Even if the handler calls signal() again as the very first thing it does, that may be too late.

Various Unix flavors played a bit with the semantics to improve on this situation. Some block signals as long as the process has not returned from the handler. The BSD solution was to invent a new system call, sigaction() where one can precisely specify the desired behavior. Today signal() must be regarded as deprecated - not to be used in serious applications.

Blocking signals

Each process has a list (bitmask) of currently blocked signals. When a signal is blocked, it is not delivered (that is, no signal handling routine is called), but remains pending.

The sigprocmask() system call serves to change the list of blocked signals.

The sigpending() system call reveals what signals are (blocked and) pending.

The sigsuspend() system call suspends the calling process until a specified signal is received.

When a signal is blocked, it remains pending, even when otherwise the process would ignore it.

wait and SIGCHLD

When a process forks off a child to perform some task, it is probably interested in how things went. Upon exit, the child leaves an exit status that should be returned to the parent. So, when the child finishes it becomes a zombie - a process that is dead already but does not disappear yet because it has not yet reported its exit status.

Whenever something interesting happens to the child (it exits, crashes, traps, stops, continues), and in particular when it dies, the parent is sent a SIGCHLD signal.

The parent can use the system call wait() or waitpid() or so, there are a few variations, to learn about the status of its stopped or deceased children. In the case of a deceased child, as soon as a status has been reported, the zombie vanishes.

If the parent is not interested it can say so explicitly (before the fork) using

```
signal(SIGCHLD, SIG_IGN);
```

or

```
struct sigaction act;
act.sa_handler = something;
act.sa_flags = SA_NOCLDWAIT;
sigaction (SIGCHLD, &act, NULL);
```

and as a result it will not hear about deceased children, and children will not be transformed into zombies. Note that the default action for SIGCHLD is to ignore this signal; nevertheless signal(SIGCHLD, SIG_IGN) has effect, namely that of preventing the transformation of children into zombies. In this situation, if the parent does a wait(), this call will return only when all children have exited, and then returns -1 with errno set to ECHILD.

It depends on the Unix flavor whether SIGCHLD is sent when SA_NOCLDWAIT was set. After act.sa_flags = SA_NOCLDSTOP no SIGCHLD is sent when children stop or stopped children continue.

If the parent exits before the child, then the child is reparented to init, process 1, and this process will reap its status.

Returning from a signal handler

When the program was interrupted by a signal, its status (including all integer and floating point registers) was saved, to be restored just before execution continues at the point of interruption.

This means that the return from the signal handler is more complicated than an arbitrary procedure return - the saved state must be restored.

To this end, the kernel arranges that the return from the signal handler causes a jump to a short code sequence (sometimes called trampoline) that executes a sigreturn() system call. This system call takes care of everything.

ALGORITHM:

PROGRAM (With output and Comments):

CONCLUSION (At least three points):

ASSIGNMENTS:

1. Enlist the system calls related to Signals.

REFERENCES:

1. “Beginning Linux Programming” by Neil Mathew and Richard Stones, Wrox Publications.
2. http://win.tue.nl/

Tushar B Kute
http://tusharkute.com