LINUX Signals

A feature of LINUX programming is the idea of sending and receiving signals. A signal is a kind of (usually software) interrupt, used to announce asynchronous events to a process.

There is a limited list of possible signals; we do not invent our own. (There might be 64 signals, for instance.) The name of a LINUX signal begins with "SIG". Although signals are numbered, we normally refer to them by their names. For example:

- SIGINT is a signal generated when a user presses Control-C. This will terminate the program from the terminal.
- SIGALRM is generated when the timer set by the alarm function goes off.
- SIGABRT is generated when a process executes the abort function.
- SIGSTOP tells LINUX to pause a process to be resumed later.
- SIGCONT tells LINUX to resume the processed paused earlier.
- SIGSEGV is sent to a process when it has a segmentation fault.
- SIGKILL is sent to a process to cause it to terminate at once.

What happens when a signal occurs?

When the signal occurs, the process has to handle it. There are three cases:

- Ignore it. Many signals can be and are ignored, but not all. Hardware exceptions such as "divide by 0" (with integers) cannot be ignored successfully and some signals such as SIGKILL cannot be ignored at all.
- Catch and handle the exception. The process has a function to be executed if and when the exception occurs. The function may terminate the program gracefully or it may handle it without terminating the program.
- Let the default action apply. Every signal has a default action. The default may be:
  - ignore
  - terminate
  - terminate and dump core
  - stop or pause the program
  - resume a program paused earlier

Each signal has a current "disposition" which indicates what action will be the default; an

additional option is to have a programmer-defined function to serve as the signal handler.

An example of the use of signals is the use of the waitpid() function. It puts the calling process in a wait state (action = STOP) until the child process indicated has a change of status, which will be reported by a SIGCHILD signal (action = resume). By default, waitpid() expects the child to terminate, but there are ways to change this so other changes of status can be handled.

The number of possible signals is limited. The first 31 signals are standardized in LINUX; all have names starting with SIG. Some are from POSIX.

#	Signal Name	Default Action	Comment	POSIX
1	SIGHUP	Terminate	Hang up controlling terminal or process	Yes
2	SIGINT	Terminate	Interrupt from keyboard, Control-C	Yes
3	SIGQUIT	Dump	Quit from keyboard, Control-\	Yes
4	SIGILL	Dump	Illegal instruction	Yes
5	SIGTRAP	Dump	Breakpoint for debugging	No
6	SIGABRT	Dump	Abnormal termination	Yes
6	SIGIOT	Dump	Equivalent to SIGABRT	No
7	SIGBUS	Dump	Bus error	No
8	SIGFPE	Dump	Floating-point exception	Yes
9	SIGKILL	Terminate	Forced-process termination	Yes
10	SIGUSR1	Terminate	Available to processes	Yes
11	SIGSEGV	Dump	Invalid memory reference	Yes
12	SIGUSR2	Terminate	Available to processes	Yes
13	SIGPIPE	Terminate	Write to pipe with no readers	Yes
14	SIGALRM	Terminate	Real-timer clock	Yes
15	SIGTERM	Terminate	Process termination	Yes
16	SIGSTKFLT	Terminate	Coprocessor stack error	No
17	SIGCHLD	Ignore	Child process stopped or terminated or got a signal if traced	Yes
18	SIGCONT	Continue	Resume execution, if stopped	Yes
19	SIGSTOP	Stop	Stop process execution, Ctrl-Z	Yes
20	SIGTSTP	Stop	Stop process issued from tty	Yes
21	SIGTTIN	Stop	Background process requires input	Yes
22	SIGTTOU	Stop	Background process requires output	Yes
23	SIGURG	Ignore	Urgent condition on socket	No
24	SIGXCPU	Dump	CPU time limit exceeded	No
25	SIGXFSZ	Dump	File size limit exceeded	No
26	SIGVTALRM	Terminate	Virtual timer clock	No
27	SIGPROF	Terminate	Profile timer clock	No
28	SIGWINCH	Ignore	Window resizing	No
29	SIGIO	Terminate	I/O now possible	No
29	SIGPOLL	Terminate	Equivalent to SIGIO	No
30	SIGPWR	Terminate	Power supply failure	No
31	SIGSYS	Dump	Bad system call	No
31	SIGUNUSED	Dump	Equivalent to SIGSYS	No

Notice SIGUSR1 and SIGUSR2. These are available for customized use. For each the default action is Terminate, but the programmer can change that. A programmer can use these to

provide an absolutely minimal amount of communication, i.e., "something happened", between processes.

When a process uses the fork() function), the child inherits a copy of the signal dispositions of its parent. If the child then uses one of the exec() functions, the status of all signals is reset to either ignore or the default, regardless of ths situation in the parent process.

A process can change the disposition of a signal using the sigaction() function:

```
int sigaction(int S, const struct sigaction * Act, struct sigaction * OldAct)
```

This changes the action taken by a process when it receives a specific signal S (any signal except SIGKILL and SIGSTOP). If Act is non-null, the new action is installed from Act. If OldAct is non-null, the previous action is stored int it. Here sigaction contains the address of the handler and some other data such as a mask of signals that will be blocked during the execution of the handler. Writing a handler requires some care, as your program is being interrupted and you don't know at which point.

An alternative is the signal() function, easier to use but less standardized than sigaction().

If we do want to have our own function to handle a signal, it might look like:

```
void handler(int S)
```

If we have multiple threads, the disposition of a signal is the same for all of the threads.

How does a process send a signal? Use one of these:

- int raise(getpid(), int S)

  This sends a signal S to the calling thread (hence getpid()). If a handler is called, raise() returns after the handler returns. The return value is 0 for success and -1 for failure.

- int kill(pid_t PID, int S)

  This sends a signal S to a specified process (if PID >0) or to all members of a specified process group (if PID = 0) or to all processes on the system (if PID = -1). The return value is 0 if at least one signal was set and -1 for errors.

- int killpg(int PGroup, int S)

  This sends a signal S to all members of a specified process group whose process group ID is PGroup. The return value is 0 if at least one signal was set and -1 for errors.

- int pthread_kill(pthread_t TID, int S)

  This sends a signal S to a specified thread with thread ID TID in the same process as the

caller. (Internally this uses tqkill().) The return value is 0 for success and -1 for errors.

- int tgkill(int TGIG, int TID, int S)

  This sends a signal S to a specified thread with thread ID = TID in a specified thread group with ID = TGID. The return value is 0 for success and -1 for errors.

- int siqqueue(pid_t PID, int S, const union sigval V)

  This sends a real-time signal S and some data (indicated by V) to a specified process with process ID = PID. (Here the union sigval contains either an integer or a void pointer; pass a number or an address.)

A process can be made to wait until a signal is caught, using:

- int pause()

  This suspends execution until any signal is caught. It returns only when a signal is caught and handled (rather than the process terminating), in which case the return value is -1.

- int sigsuspend(const sigset_t *Mask)

  This temporarily changes the signal mask to *Mask and suspends execution until one of the unmasked signals is caught. The return value is always -1.

Also remember that we have the wait() and waitpid() functions. These will wait for a change in the state of any child process or a specified child process. The state change might be that the child terminated, the child was stopped by a signal, or the child was resumed by a signal.

What is the signal mask?

A signal may be "blocked" so it will not be delivered until it is is "unblocked". While the signal is waiting between being generated and being delivered, it is "pending".

Each thread in a process has its own signal mask which lists the signals that thread is currently blocking. A thread can modify its signal mask using pthread_sigmask(), and a (single-threaded) process can do so using sigprocmask().

If a process creates a child using fork(), the child inherits a copy of its parent's signal mask, and this is preserved even if one of the exec() functions is then called.

Example C Program to Catch a Signal (from a web page)

Most of the Linux users use the key combination Ctr+C to terminate processes in Linux.

Have you ever thought of what goes behind this? Well, whenever Ctrl+C is pressed, a signal SIGINT is sent to the process. The default action of this signal is to terminate the process. But this signal can also be handled. The following code demonstrates this:

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

void sig_handler(int signo)
{
  if (signo == SIGINT)
    printf("received SIGINT\n");
}

int main(void)
{
  if (signal(SIGINT, sig_handler) == SIG_ERR)
  printf("\ncan't catch SIGINT\n");
  // A long long wait so that we can easily issue a signal to this
  // process
  while(1)
    sleep(1);
  return 0;
}
```

In the code above, we have simulated a long running process using an infinite while loop.

A function sig_handler is used as a signal handler. This function is "registered" to the kernel by passing it as the second argument of the system call signal in the main() function. The first argument to the function signal is the signal we intend the signal handler to handle which is SIGINT in this case.

On a side note, the use of function sleep(1) has a reason behind. This function has been used in the while loop so that while loop executes after some time (i.e. one second in this case). This becomes important because otherwise an infinite while loop running wildly may consume most of the CPU making the computer very very slow.

Anyway, coming back, when the process is run and we try to terminate the process using Control-C, we get:

```
$ ./sigfunc
^Creceived SIGINT
^Creceived SIGINT
^Creceived SIGINT
^Creceived SIGINT
^Creceived SIGINT
^Creceived SIGINT
^Creceived SIGINT
```

We see in the above output that we tried the key combination Control+C several times but

each time the process did not terminate. This is because the signal was handled in the code and this was confirmed from the print we got on each line.

Notice that after the signal handler executes, the program continues inside the loop, that is, where it was before the signal. This is what should happen if the signal handler does not end the program.

Actually, there is a problem with the above example. The code for a signal handler should be reentrant. Why? When the signal handler is invoked, the process waits while the handler is being executed. What if another signal occurs? The second signal may be handled at once and then the handler for the first signal will resume. We could have a stack of handlers.

Reentrant code can be interrrupted at any point and resume later without losing data. Thus, for instance, reentrant code cannot use malloc() (to allocate memory dynamically) as malloc() is not reentrant.

The example given above is not reentrant because of the use of printf(). A better idea is illustrated here:

```
#include <stdlib.h>
#include <signal.h>
#include <stdio.h>
#include <string.h>
#include <unistd.h>

// The got_signal variable serves as a flag
// to indicate the reception of a signal.
static volatile sig_atomic_t got_signal = 0;

static void my_sig_handler(int signo)
{
 got_signal = 1;
}

int main()
{
 struct sigaction sa;

 memset(&sa, 0, sizeof(struct sigaction));
 sa.sa_handler = &my_sig_handler;
 if (sigaction(SIGINT, &sa, NULL) == -1)
 {
   perror("sigaction");
   return EXIT_FAILURE;
 }

 while(1)
 {
  if (got_signal)
  {
   got_signal = 0;
   printf("Received interrupt signal SIGINT!\n");
  }
```

```
  printf("Doing useful stuff...\n");
  sleep(1);
 }
 return EXIT_SUCCESS;
}
```

Notice that we can exit from this program with Control-Z or Control-\.

All the signal handler does in this example is change the value of one atomic variable.