COMP421
Unix Environment for Programmers
Lecture 17: Signals

Jeff Wiegley, Ph.D.
Computer Science

jeffw@csun.edu

10/17/2005

> ‘‘Ships that pass in the night, and speak each other in passing, only a signal shown, and a distant voice in the darkness; So on the ocean of life, we pass and speak one another, only a look and a voice, then darkness again and a silence.’’
>
> – *Henry Wadsworth Longfellow*

Signals:

"Signals" present a simple mechanism for processes to communicate with each other.

They are software interrupts.

They function like interrupt vectors (which will be familiar to those with some background in electrical engineering or micro-controllers.)

Signal names.

Every signal has a name (which maps to a number). Common names are `SIGHUP`, `SIGKILL`, `SIGTERM` and `SIGUSR1`

There is no difference between signals. They can all do the same thing.[a]

The variety allows a program to tailer its behavior depending on which signal it received.

[a] except SIGKILL and SIGSTOP, which cannot be caught or ignored.

What causes signals to be sent:

Signals are caused by a variety of events:

- Pressing CTRL-D in many actions causes a SIGINT to be sent to a process. (This is what the shell really does when you press CTRL-D).
- Hardware exceptions, such as division by zero and invalid memory references (SIGSEGV), also cause signals.
- The `kill` command from the shell can be used to send any signal to any process. This is often used to send SIGHUP to daemons to get them to reload their configuration.
- the `kill()` function can be used inside of programs to allow one process to send a signal to another process.
- Software signals can also occur. For window size changes (SIGWINCH), out-of-band data on a network connection (SIGURG) or if one of a pipe is absent (SIGPIPE). Ctrl-D Thread synchronization is based on the concepts of:

What can we do with signals:

- We can ignore them. (except for KILL and STOP)
- We can allow there default handler to execute.
- We can specify a new function to run when a signal is received. (except for KILL and STOP)

If we change the handler then, when a matching signal is received, the program will immediately interrupt what is doing and run the signal handler function instead.

When the signal handler returns the programs picks up where it left off when it was interrupted.

Changing the handler:

The simplest method for changing the handler is:

```
#include <signal.h>

typedef void (*sighandler_t)(int);

sighandler_t signal(int signum, sighandler_t handler);
```

Which is basically a simple function that takes two arguments:

1. The number of the signal for which you want to change behavior.
2. A function pointer to the function that you want run when the signal is received. SIG_IGN and SIG_DFL can also be passed to ignore the signal or choose default handling.

An example:

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

void myhandler(int signo)
{
  fprintf(stderr,"Damn it! I refuse to quit!!");
}

void updateconf(int signo)
{
  fprintf(stderr,"reloading configuration file");
}

int main(int argc, char *argv[])
{
  signal(SIGINT, myhandler);
  signal(SIGHUP, updateconf);
  while (1); /* infinite loop */
}
```

Exec and signals:

When `exec()` is called to change a process. (such as after `fork()`) exec does the following:

- Any signals that were being ignored continue to be ignored.
- All other signals are set to their default action.[a]

[a]Think about it. Since the code segment is being entirely replaced by a new code segment, would the process be able to call a function that was only present in the old code segment? No!

Problems with earlier systems:

In early systems, signals where “unreliable”.

This was due to the fact the signal handler was reset to SIG_IGN each time the signal was received.

Classic solution was to re-enable the signal handler as the first or last step in the signal handling function itself.

This meant that there was a small window of time after a first occurrence of a signal was received when a second instance of the signal could occur before the signal handler was reset and the default handler would execute. For signals such as SIGTERM this meant the process died when it shouldn’t have. (The default action for SIGTERM is to kill the process.)

Another problem:

There are times when you want to ignore a signal (during a critical calculation, for instance)

But you still want to know that the signal occurred so that you can process it after the critical computation is completed.

Early systems only allowed you to suspend signals by simply ignoring them. This has no memory that signals ever occurred and therefore aren't suitable to solve this problem.

Signals can interrupt System calls:

Remember that `read()` is a blocking system call?[a]

Q: Well guess what happens if the process receives a signal while `read()` is blocked?

A: `read()` unblocks! (and more specifically returns -1 indicating that an error occurred.)

So, as programmers we have to be very careful when using system calls in conjunction with signals that we very carefully check the return status of the system call.

In this case, `read()` returned -1 AND set `errno` to `EINTR`. So before assuming that `read()` failed because of some fatal error we have to check `errno` first.

[a] heaven help you on the final if you don't.

WARNING!!!!:

Signal handler should not make use of any NON-reentrant functions.

This includes `malloc()` and `free()`, all system calls that utilize these and all other non-reentrant system calls.

Figure 10.4 of the book lists the known reentrant system calls that are safe to use in signal handlers.

Some signal functions:

There is a lot of code based on earlier, unreliable, signals.

POSIX provides a suite of functions that implement reliable signals.

- `kill(int proc, int signum)`, send a signal to another process.
- `raise(int signum)`, send a signal to myself.
- `alarm(unsigned int seconds)`, set a software timer that will deliver a SIGALRM to calling process in the future.
- `pause()`, suspend calling function until a signal occurs.

Signal sets:

POSIX provides a suite of functions that implement reliable signals.

First we need to know about signal sets:

- `int sigemptyset(sigset_t *set)` clears a signal "set".
- `int sigfillset(sigset_t *set)` clears a signal
- `int sigemptyset(sigset_t *set)` initializes the signal set given by set to empty, with all signals excluded from the set.
- `int sigfillset(sigset_t *set)` initializes set to full, including all signals.
- `int sigaddset(sigset_t *set, int signum)` add signal signum from set.
- `int sigdelset(sigset_t *set, int signum)` delete signal signum from set.
- `int sigismember(const sigset_t *set, int signum)` tests whether signum is a member of set.

Reliable Signals:

- `sigprocmask()`, change which signals are blocked or delivered to the calling process.
- `sigpending()`, determine the set of signals currently blocked but pending delivery.
- `sigaction()`, examine or modify the actions associated with a particular signal.
- `sigsuspend()`, Change process mask and suspend calling process until a signal is received.

Blocking signals:

`int sigprocmask(int how, const sigset_t *set, sigset_t *oldset)`

`how` is one of `SIG_BLOCK`, `SIG_UNBLOCK` or `SIG_SETMASK`.

if `oldset` is not NULL then the previous signal mask is stored in `oldset`.

if `set` is not NULL then the set of signals specified is either blocked or unblocked. (or if `how==SIG_SETMASK` then the set of block signals is set equal to `set`.

Unblocking a signal causes any pending signals to be delivered. However, multiple generations of a signal while blocked do not queue up. When unblocked only a single signal is delivered.[a]

[a] operating system dependent.

Checking for pending signals:

```
int sigpending(sigset_t *set);
```

The set returned consists of the signals that have been raised while blocked.

Changing signal handlers:

```
int sigaction(int signum,
              const struct sigaction *act,
              struct sigaction *oldact)

      struct sigaction {
                  void (*sa_handler)(int);
                  void (*sa_sigaction)(int, siginfo_t *, void *);
                  sigset_t sa_mask;
                  int sa_flags;
                  void (*sa_restorer)(void);
              }
```

`sa_mask` provides a set of signals which are blocked during execution of the `sa_handler`.

`sa_flags` can be used to modify behavior of the signal handler. `SA_RESETHAND`, for instance, will cause the signal's handler to be the default action after the handler runs (=early implementations).

Conclusion.

In conclusion, the reliable signal routines should be used for all new applications.