Signals 3: Signal Handlers

1. Overview
2. System Calls
3. **Signal Handlers**
   - **signal handlers syntax**
   - **examples**
   - **passing data: global variables**
   - **siginfo handlers**
4. Synchronization
5. Signals and fork, exec, Pthreads
6. Advanced Topics
7. Signal Handler Issues

Registering a Signal Handler

As note previously, `signal()` should be used *only to set default or ignore disposition*.

Setting signal disposition to be *caught* by a *signal handler* should *always be done with* the newer `sigaction()`.

(This is often termed *establishing* or *registering* a handler.)

`sigaction()` requires creating an appropriate sigaction struct:

```
struct sigaction act;
memset(&act,0,sizeof(act));
act.sa_handler = sigint_handler;  //Name of handler function
sigemptyset(&act.sa_mask);
sigaddset(&act.sa_mask,SIGCHLD);
act.sa_flags = SA_RESTART;

sigaction(SIGINT,&act,NULL)
```

Signal Handlers Syntax

What is the *syntax* for defining a *signal handler* function?

`signal()`'s syntax is commonly shown using a `typedef` for the signal handler syntax to simplify things.

The typedef used with `signal()` specifies the signal handler as: `void (*sighandler_t)(int)`:

- (**function*) denotes a function (pointer)
- `(int)` indicates the function takes a single, `int` argument
- `void` indicates the function has a `void` return type

Thus, to be acceptable as a *signal handler*, a function must take a single `int` argument and have `void` return type.

Signal Handlers Syntax (contd.)

The sigaction struct does not use a `typedef` for the handler element, but species the same syntax for handler functions:

```
void (*sa_handler)(int)
```

Note that without using a `typedef`, `signal()` could be defined as:

```
void (*oldhandler)(int) signal(int signum, void (*newhandler)(int))
```

Signal Handler Examples

Example signal handler function for use with SIGINT:

```
void sigint_handler(int signal)
{
  //Beep terminal instead of terminating process on ^-c:
  write(1,"\a",1);  //'\a' is the ASCII BEL character
  return;
}
```

Example signal handler function for use with SIGCHLD:

```
void sigchld_handler(int signal)
{
  //Collect all terminated children:
  while (waitpid(-1,NULL,WNOHANG) > 0);
  errno=0;  //in case waitpid() set it
  return;
}
```

Passing Data to Handlers

The only information passed to basic signal handlers is the signal that invoked the handler (but see "**siginfo handlers**" below).

Obviously this is not useful when the handler is registered for a single signal.

However, the same handler function may be registered for multiple signals, so the handler may need to check which signal invoked it to determine what to do.

Often, it is necessary for the program that setup a handler to be able to pass data to or from a handler.

Because of signal handlers' syntax, this requires **global variables**.

(In C, *global variables* are variables declared outside of `main` or any functions, giving them **global scope**.)

Example of Passing Data via Global Variable

Terminate loop in program when user types ^-c:

```
int done = 0;  //Global flag variable

int main()
{
  ...use sigaction() to register sigint_handler() as handler for SIGINT...
  ...
  while (!done) {
    ...do stuff...
  }
  return EXIT_SUCCESS;
}

void sigint_handler(int signal)
{
  done=1;
  return;
}
```

Siginfo Handlers

Standard signal handlers have to have prototypes like:

```
void sa_handler(int signo)
```

`sigaction()` can be used to setup such handlers via the sigaction struct's `sa_handler` field.

However, `sigaction()` can instead setup "**siginfo handlers**."

These handlers receive additional information about the signal context, and have prototypes:

```
void sa_sigaction(int signo, siginfo_t *si, void *cntxt)
```

To register such a handler:

- the `SA_SIGINFO` flag must be set in the sigaction struct
- the handler function is set in the `sa_sigaction` field
- no handler is set using the `sa_handler` field

Siginfo Handlers (contd.)

`siginfo_t` is a struct with additional info about signal context.

It is somewhat complex, since it contains a **union** that defines different fields depending upon the signal it results from.

Key fixed fields:

- `int si_signo` – the signal (number)
- `int si_errno` – errno value if nonzero
- `int si_code` – for some signals, code that identifies event

An important aspect of siginfo handlers is the ability for limited data to be *passed to the signal handler* when using `sigqueue()` (see below) to send a signal to a process.

Siginfo Handlers (contd.)

This involves the `siginfo_t` struct's `si_value` field, a *union*:

```
union {
  int   sival_int;
  void *sival_ptr;
}
```

Thus, we can pass either an integer value (in `si_value.sival_int` or a pointer value (in `si_value.sival_ptr`).

(Note that pointers will be valid only if a process uses `sigqueue()` to send a signal to *itself!*)

This allows data to be passed to handlers *without* the use of **global variables**.

That can be particularly useful with **multithreaded programs** (since globals are *shared* among threads).