

CSIS0230A Principle of Operating Systems (Class A)

Notes for Tutorial 8

Signals

In this tutorial, you will do some experiments on signals. We will use the POSIX signal interface to obtain inner information about the signal.

1. Sending signals

A process can send a signal to another process by `kill()`. It is defined in `<signal.h>` as `int kill(pid_t pid, int sig)`. The first argument specifies the process (or process group in case `pid<0`) receiving this signal. The second argument specifies the signal type to be sent. It returns 0 on success, and -1 on error. To send a signal to itself, one can use `int raise(int sig)` as well.

Each system has a predefined fixed set of signal types, and you can find symbolic names of each signal type in `<asm/signal.h>`. Some examples are:

```
#define SIGUSR1    10
#define SIGSEGV   11
#define SIGCHLD   17
#define SIGSTOP   19
```

Most signals can be generated by the OS (as well as generated by `kill()`). For example, when the OS terminates a process, `SIGCHLD` is sent to its parent; and when the page fault handler determines that a process incorrectly accesses a memory location, it generates `SIGSEGV`. Other signals, like `SIGSTOP` and `SIGUSR1`, are only sent by user functions.

2. What to do on signal reception?

When a process receives a signal, some **action** is taken. The default action depends on the type of signal received. For example, upon receiving `SIGUSR1`, `SIGSEGV`, `SIGCHLD` and `SIGSTOP`, the default actions are to terminate the process, terminate the process, ignore the signal and stop the process (until some process sends it `SIGCONT`) respectively. These are documented in the man page `signal(7)`.

For most types of signals (that is, except `SIGSTOP` and `SIGKILL`), the process can modify the default behaviour by calling `sigaction()`, defined in `<signal.h>`:

```
int sigaction(int signum, const struct sigaction *act,
              const struct sigaction *oact);
```

This retrieves the previous signal action to `oact`, and sets the new signal action to `act`. If either argument is `NULL`, the corresponding retrieval or setting is not performed.

The signal action structure has three members of interest. The member `sa_flags` specifies some flags of the action, e.g., whether to reset the signal handler to default after the signal is caught, etc. The `sa_handler` member defines a function to be called upon receiving the signal, unless the `sa_flags` member contains `SA_SIGINFO`, when the `sa_sigaction` member is used instead. The difference is that a `sa_handler` is called only with the signal number, while `sa_sigaction` is called with other information related to the signal.

3. The signal handler

A signal handler is a function of the following prototype:

```
void handler(int signum);
```

Here *signum* is the type of signal (like *SIGINT*) which triggered the signal handler. There are two special handlers defined: *SIG_DFL* specifies that the default (as described in `signal(7)`) is used, while *SIG_IGN* specifies that the signal should be ignored.

When writing signal handlers, one should be aware that many functions should not be used, because they use shared data structures. For example, the C library function *getlogin()*, which gets the login name of the current user, uses a static area of the process to write the result, and the memory might be used just before the signal handler is called. In that case the memory will get corrupted. The C library provides alternative functions like *getlogin_r()* to perform the same functionality without using static memory. Such functions should always be used instead.

4. Extended information: sigaction handlers

A signal can have associated information. Such information can be retrieved using a sigaction. A sigaction handler has the following prototype:

```
void handler(int signum, struct siginfo_t *info, void *context);
```

Compared with a regular handler, it contains two more fields: *info* and *context*. Here *context* is a pointer to a structure that contains information about the context (i.e., register values) of the process when the signal is sent. The more interesting parameter *info* contains information about why the signal get sent. The *siginfo_t* structure is defined like this:

```
siginfo_t {
    int          si_signo; /* Signal number */
    int          si_errno; /* An errno value */
    int          si_code; /* Signal code */
    pid_t        si_pid; /* Sending process ID */
    uid_t        si_uid; /* Real user ID of sending process */
    int          si_status; /* Exit value or signal */
    clock_t      si_utime; /* User time consumed */
    clock_t      si_stime; /* System time consumed */
    void *       si_addr; /* Memory location which caused fault */
    ...
}
```

The most important of these is the things after *si_code*. The field *si_code* stores information about why a signal is sent: whether it is due to a user calling *kill* (*SI_USER*), or due to the kernel itself. If the signal is sent by the user, *si_pid* and *si_uid* stores the PID and UID of the sender. If the signal is *SIGCHLD*, then *si_status* is set to the exit status, and *si_utime* and *si_stime* are set to the user and system time consumed by the process. If the signal is *SIGILL* or *SIGSEGV*, *si_addr* will hold the faulting address that caused the error. The complete list of *si_code* values and *siginfo_t* fields can be found in the manpage of `sigaction(2)`.