

CSIS0230A Principles of Operating Systems (Class A)

Notes for Tutorial 6

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 type of signal to be sent. If succeeded, it returns 0. To send a signal to oneself, `int raise(int sig)` may also be used.

Signal types are small integers with predefined meaning. You can find their symbolic names in `<asm/signal.h>`. E.g., `SIGUSR1`, `SIGSEGV`, `SIGCHLD` and `SIGSTOP` are defined to be 10, 11, 17 and 19 respectively. Many types of signals can be generated by the OS kernel, apart from being generated by `kill()`. E.g., when a process terminates, the kernel sends it parent `SIGCHLD`; and when a process access invalid memory location, the kernel sends it `SIGSEGV`. Some types of signals, like `SIGSTOP` and `SIGUSR1`, are only sent by user processes using `kill()`.

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, and is documented in the man page `signal(7)`. E.g., by default `SIGUSR1` and `SIGSEGV` terminates the process, `SIGCHLD` is silently ignored, and `SIGSTOP` stops the process until `SIGCONT` is received.

The process can modify the behaviour upon receiving a type of action (except `SIGSTOP` and `SIGKILL`), by using a new **action** for it. An action is specified using a `sigaction` structure:

```
struct sigaction { /* defined in <signal.h> */
    void (*sa_handler)(int);
    void (*sa_sigaction)(int, siginfo_t *, void *);
    sigset_t sa_mask;
    int sa_flags;
    void (*sa_restorer)(void);
};
```

We will use 3 members of the struct. `sa_flags` is the bitwise-or of some flags modifying the behaviour of the action, e.g., whether to reset the action to default after the signal is caught. Depending on whether or not `sa_flags` contains `SA_SIGINFO`, either `sa_sigaction` or `sa_handler` points to the function to be called upon receiving the signal. The former is more informative than the latter, as can be seen by the number of arguments in their function signatures.

To use an action, call `sigaction()`, with the following prototype:

```
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.

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)`) should be used, while *SIG_IGN* specifies that the signal should be ignored. A simple signal handler looks like this:

```
void my_sigchld_handler(int signum) { /* Call wait to avoid zombie */  
    int old_errno = errno, status;  
    wait(&status); /* ignore errors */  
    errno = old_errno;  
}
```

It can be tricky to write correct signal handlers. The problem is similar to interrupt handlers: they **occur at any time** (any time when the process is in user mode). E.g., suppose your signal handler calls *malloc*. What will happen if, when the signal occurs, the process is calling *malloc* itself? The result is chaotic: *malloc* probably is in the middle of modifying a linked list, and the *malloc* within the signal handler will see an inconsistent linked list. The result in general is a segmentation fault which is very difficult to locate.¹

So a signal handler should be mostly **reentrant**, i.e., it should have minimal effect to the remainder of the code. Functions like *malloc* should not be used in signal handler at all. Many functions modify global variables, and if those global variables are used outside the signal handler, the handler must either refrain from calling them, or be careful to save and restore the variables. The above example shows how this is done to system calls, which modify the global variable *errno*.

4. Getting more information: sigaction handlers

As discussed in Section 2, one can choose to use a **sigaction** instead of a handler. This reveals more information of the signal. E.g., a sigaction is informed about the reason why the signal is sent, and if the signal is sent by a process using *kill* or *raise* it is also informed about the process which sends the signal. A sigaction handler has the following prototype:

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

Compared with a regular handler, a sigaction receives two more arguments: *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. This is seldom used, as it is highly implementation dependent.² On the other hand, *info* is system independent, containing extended information about the signal. The *siginfo_t* structure is defined like this:

¹This is a special synchronization problem, involving two independent “threads of execution”—the main program and the signal handler. However, we cannot solve the problem using a semaphore, as the main program will not continue at all until the signal handler completes (what will happen if we use semaphore?). The *malloc* example can be solved if we can temporarily prevent signal handlers from being called, during the time when the main program calls *malloc*, reenabling signal handling afterwards. This can be achieved using *sigprocmask()*.

²For Linux, it is actually a **struct** *ucontext**. If you decide to manipulate it, see the man page `getcontext(2)` and the header file `<asm/sigcontext>`.

```
typedef struct siginfo {  
    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 */  
    ...  
} siginfo_t;
```

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)`.