Advanced Programming in the UNIX Environment — *Signals*

Hop Lee
hoplee@bupt.edu.cn

Contents

1 Signals Concepts

- **Signals** are software interrupts. They provide a way of handling asynchronous events.
- Signals have been provided since the early versions of UNIX. But some early signal models are not reliable.
- Version 7 had 15 different signals; SVR4 and 4.3+BSD both have 31 different signals. FreeBSD 8.0 supports 32, Mac OS X 10.6.8 and Linux 3.2.0 each support 31, whereas Solaris 10 sopports 40 signals.
- Every signal has a name beginning with the three characters **SIG**.
- These name are all defined by positive integer constants in the header `signal.h`
- Numerous conditions can generate a signal.
  - The terminal-generated signals occur when users press certain terminal key.
  - Hardware exceptions generate signals.
  - The `kill`(2) function allows a process to send any signal to another process or process group under some kind of limitations.
  - The `kill`(1) command allows us to send signals to other processes.
  - Software conditions can generate signals when something happens that the process should be made aware of.
- There are three different things that we can tell the kernel to do when a signal occurs
  1. Ignore the signal. This works for most signals, but there are two signals that can never be ignored: `SIGKILL` and `SIGSTOP`.
  2. Catch the signal. To do this we tell the kernel to call a function of ours when ever the signal occurs.
  3. Let the default action apply. Every signal has a default action.
- Figure 10.1 on textbook lists the names of all the signals, an indication of which systems support the signal, and the default action for the signal.

2 `signal` Function

- The simplest interface to the signal features of UNIX is the `signal` function.

```
#include <signal.h>
void (*signal(int signo, void (*func)(int)))(int);
```

- The *signo* argument is just the name of the signal from Figure 10.1.
- The value of *func* is either
  1. the constant `SIG_IGN`
  2. the constant `SIG_DFL`
  3. the address of a function to be called when the signal occurs.
- The return value from `signal` is the pointer to the previous signal handler or `SIG_ERR` on error.
- Example (Figure 10.2, `signals/sigusr.c`) shows a simple signal handler that catches either of the two user-defined signals and prints the signal number.

3 Unreliable Signals

- In earlier versions of the UNIX System (such as Version 7), signals were *unreliable*. By this we mean that signals could get los. Also, a process had little control over a signal: a process could catch the signal or ignore it. Sometimes, we would like to tell the kernel to block a signal.

- One problem with these early versions was that the action for a signal was reset to its default each time the signal occurred. Example:

```
  /* my signal handling function */
  int sig_int();
  ...
  /* establish handler */
  signal(SIGINT, sig_int);
  ...
sig_int()
{
  /* reestablish handler for next time */
  signal(SIGINT, sig_int);
  ... /* process the signal ... */
}
```

- Another problem with these earlier systems was that the process was unable to turn a signal off when it didn't want the signal to occur. Example:

```
/* my signal handling function */
int sig_int();
/* set nonzero when signal occurs */
int sig_int_flag;
main()
{
  /* establish handler */
  signal(SIGINT, sig_int);
  ...
  while (sig_int_flag == 0)
  /* go to sleep, waiting for signal */
  pause();
  ...
}
sig_int()
{
  /* reestablish handler for next time */
  signal(SIGINT, sig_int);
  /* set flag for main loop to examine */
  sig_int_flag = 1;
}
```

4 Interrupted System Calls

- A characteristic of earlier UNIX systems was that if a process caught a signal while the process was blocked in a "slow" system call, the system call was interrupted. The system call returned an error and `errno` was set to EINTR. This was done under the assumption that since a signal occurred and the process caught it, there is a good chance that something has happened that should wake up the blocked system call.

- The problem with interrupted system calls is that we now have to handle the error return explicitly. Example:

```
again:
  if ((n = read(fd, buf, BUFFSIZE)) < 0) {
    if (errno == EINTR)
      /* just an interrupted system call */
      goto again;
    /* handle other errors */
  }
```

- To prevent applications from having to handle interrupted system calls, 4.2BSD introduced the automatic restarting of certain interrupted system calls. Since this caused a problem for some applications that didn't want the operation restarted if it was interrupted, 4.3BSD allowed the process to disable this feature on a per-signal basis.

- Figure 10.3 on textbook summarizes the signal functions and their semantics provided by the various implementations.

5 Reentrant Functions

- When a signal that is being caught is handled by a process, the normal sequence of instructions being executed by the process is temporarily interrupted by the signal handler.

- What if the process was in the middle of a call to a function and we call the same function from the signal handler?

- The SUS specifies the functions that are guaranteed to be safe to call from within a signal handler. These functions are reentrant and are called **async-signal safe** by the SUS. Besides being reentrant, they block any signals during operation if delivery of a signal might cause inconsistencies.

- Most of the functions that are not included in Figure 10.4 are missing because

  1. they are known to use static data structures,
  2. they call malloc or free, or
  3. they are part of the standard I/O library.

- Most implementations of the standard I/O library use global data structures in a nonreentrant way.

- Because we may modify the value of `errno` in the signal handler, therefore, as a general rule, when calling the functions listed in Figure 10.4 from a signal handler, we should save and restore errno.

- Example (Figure 10.5, `signals/reenter.c`).

6 Reliable-Signal Terminology and Semantics

- A signal is **generated** for a process (or send to a process) when the event that causes the signal occurs.

- When the signal is generated the kernel usually sets a flag of some form in the process table.

- We say that a signal is **delivered** to a process when the action for a signal is taken.

- During the time between the generation of a signal and its delivery, the signal is said to be **pending**.

- A process has the option of **blocking** the delivery of a signal. If a signal that is blocked is generated for a process, and if the action for that signal is either the default action or to catch the signal, then the signal remains pending for the process until the process either
  1. unblocks the signal
  2. changes the action to ignore the signal
- The system determines what to do with a blocked signal when the signal is delivered, not when it's generated.
- When a blocked signal is generated more than once before the process unblocks it, POSIX.1 allows the system to deliver the signal either once or more than once.
- The POSIX.1 does not specify the order in which the several signals are delivered to the process.

7 `kill` and `raise` Functions

- The `kill` function sends a signal to a process or a process group.
- The `raise` function allows a process to send a signal to itself.

```
#include <signal.h>
int kill(pid_t pid, int signo);
int raise(int signo);
```

- There are four different conditions for the *pid* argument to `kill`.

  ***pid*>0** The signal is sent to the process whose process ID is *pid*.

  ***pid*==0** The signal is sent to all processes whose process group ID equals the process group ID of the sender and for which the sender has permission to send the signal. Note that the term all processes excludes an implementation-defined set of system processes. For most UNIX systems, this set of system processes includes the kernel processes and init (pid 1).

  ***pid*<0** The signal is sent to all processes whose process group ID equals the absolute value of pid and for which the sender has permission to send the signal. Again, the set of all processes excludes certain system processes, as described earlier.

  ***pid*==**-1 The signal is sent to all processes on the system for which the sender has permission to send the signal. As before, the set of processes excludes certain system processes.

- The superuser can send a signal to any process.
- If the real or effective user ID of the sender equal the real or effective user ID of the receiver, the signal sending can be proceeded.
- One special case for the permission testing also exists: if the signal being sent is `SIGCONT`, a process can send it to any other process in the same session.
- POSIX.1 defines signal number 0 as the null signal. If the signo argument is 0, then the normal error checking is performed by `kill`, but no signal is sent. This technique is often used to determine if a specific process still exists. If we send the process the null signal and it doesn't exist, `kill` returns -1 and `errno` is set to `ESRCH`.
- If the call to `kill` causes the signal to be generated for the calling process and if the signal is not blocked, either signo or some other pending, unblocked signal is delivered to the process before `kill` returns.

8 `alarm` and `pause` Functions

- The `alarm` function allows us to set a timer that will expire at a specified time in the future.
- When the timer expires, the `SIGALRM` signal is generated.

```
#include <unistd.h>
unsigned int alarm(unsigned int seconds);
```

- The *seconds* value is the number of clock seconds in the future when the signal should be generated.
- There is only one alarm clocks per process. The return value is the second number left for the previous alarm clock.
- The `pause` function suspends the calling process until a signal is caught.

```
#include <unistd.h>
int pause(void);
```

- The only time `pause` returns is if a signal handler is executed and that handler returns. In that case, `pause` returns -1 with `errno` set to `EINTR`.
- Example for a `sleep` implementation (Figure 10.7, `signals/sleep1.c`).
- This simple implementation of `sleep` function has three problems.
  1. If the caller already has an alarm set, that alarm is erased by the first call to `alarm`. We can correct this by looking at `alarm`'s return value and wait of reset that alarm manully.
  2. We have modified the disposition for `SIGALRM`. If we're writing a function for others to call, we should save the disposition when our function is called and restore it when we're done by saving the return value from `signal` and resetting the disposition before our function returns.
  3. There is a race condition between the first call to `alarm` and the call to `pause`. On a busy system, it's possible for the alarm to go off and the signal handler to be called before we call `pause`. If that happens, the caller could suspended forever in the call to `pause`.
- Example for an improved `sleep` implementation (Figure 10.8, `signals/sleep2.c`).
- The `sleep2` function avoids the race condition from Figure 10.7. Even if the `pause` is never executed, the `sleep2` function returns when the `SIGALRM` occurs.
- There is, however, another subtle problem with the `sleep2` function that involves its interaction with other signals. If the `SIGALRM` interrupts some other signal handler, then when we call `longjmp`, we abort the other signal handler.
- Example (Figure 10.9, `signals/tsleep2.c`) shows the problem with `sleep2`.
- A common use for `alarm`, in addition to implementing the `sleep` function, is to put an upper time limit on operations that can block.
- Example (Figure 10.10. `signals/read1.c`) shows a typical scenario. But this program has two problems.
  1. The program in Figure 10.10 has one of the same flaws that we described in Figure 10.7: a race condition between the first call to alarm and the call to read.
  2. If system calls are automatically restarted, the `read` is not interrupted when the `SIGALRM` signal handler returns. In this case, the timeout does nothing.
- An improved implementation (Figure 10.11, `signals/read2.c`) works as expected, regardless of whether the system restarts interrupted system calls. However, that we still have the problem of interactions with other signal handlers, as in Figure 10.8.

9 Signal Sets

- We need a data type to represent multiple signals — a **signal set**. We'll use this with functions such as `sigprocmask` to tell the kernel not to allow any of the signals in the set to occur.
- POSIX.1 defines the data type `sigset_t` to contain a signal set and five functions to manipulate signal sets:

```
#include <signal.h>
int sigemptyset(sigset_t *set);
int sigfillset(sigset_t *set);
int sigaddset(sigset_t *set, int signo);
int sigdelset(sigset_t *set, int signo);
int sigismember(const sigset_t *set, int signo);
```

- The function `sigemptyset` initializes the signal set pointed to so that all signals are excluded.
- The function `sigfillset` initializes the signal set so that all signals are included.
- We can add and delete specific signals in the set by function `sigaddset` and `sigdelset`.
- The function `sigismember` returns 1 if true, 0 if false, -1 on error.
- Sample implementations (Figure 10.12, `signals/setops.c`).

10 `sigprocmask` Function

- A process can examine or change its signal mask by calling the `sigprocmask` function.

```
#include <signal.h>
int sigprocmask(int how, const sigset_t *restrict set,
                sigset_t *restrict oset);
```

- If there are any pending, unblocked signals after the call to `sigprocmask`, at least one of these signals is delivered to the process before `sigprocmask` returns.
- Example (Figure 10.14, `lib/prmask.c`).

11 `sigpending` Function

- `sigpending` returns the set of signals that are blocked from delivery and currently pending for the calling process.

```
#include <signal.h>
int sigpending(sigset_t *set);
```

- Example (Figure 10.15, `signals/critical.c`).

12 sigaction Function

- The sigaction function allows us to examine or modify the action associated with a particular signal. This function supersedes the signal function from earlier release of UNIX.

```
#include <signal.h>
int sigaction(int signo,
      const struct sigaction *restrict act,
      struct sigaction *restrict oact);
struct sigaction {
  void (*sa_handler)(int); /* addr of signal handler, */
  /* or SIG_IGN, or SIG_DFL */
  sigset_t sa_mask; /* additional signals to block */
  int sa_flags; /* signal options, Figure 10.16 */
  /* alternate handler */
  void (*sa_sigaction)(int, siginfo_t *, void *);
};
```

- When changing the action for a signal to a signal-catching function then the second member of second argument specifies a set of signals that are added to the signal mask of the process before the signal-catching function is called.
- If and when the signal-catching function returns, the signal mask of the process is reset to its previous value.
- We are guaranteed that whenever we are processing a given signal, another occurrence of that same signal is blocked until we're finished processing the first occurrence.
- Once we install an action for a given signal, that action remains installed until we explicitly change it by calling sigaction.
- Example (Figure 10.18, lib/signal.c) shows a reliable version of the signal function.
- Another example (Figure 10.19, lib/signalintr.c) shows a version of the signal function that tries to prevent any interrupted system calls from being restarted.

13 sigsetjmp and siglongjmp Functions

- There is a problem in calling longjmp. When a signal is caught, the signal-catching function is entered with the current signal automatically being added to the signal mask of the process.
- If we longjmp out of the signal handler, what happens to the signal mask for the process?
- POSIX.1 provide two functions sigsetjmp and siglongjmp to solve this problem.

```
#include <setjmp.h>
int sigsetjmp(sigjmp_buf env, int savemask);
void siglongjmp(sigjmp_buf env, int val);
```

- The sigsetjmp returns 0 if called directly, nonzero if returning from a call to siglongjmp.
- Example (Figure 10.20, signals/mask.c) illustrates the use of the sigsetjmp and siglongjmp functions.

14 sigsuspend Function

- If we want to unblock a signal and then **pause**, waiting for the previously blocked signal to occur, we can use `sigsuspend` function.
- This function is an atom function of unblock a signal and put the process to sleep.

```
#include <signal.h>
int sigsuspend(const sigset_t *sigmask);
```

- The signal mask of the process is set to the value pointed to by *sigmask*. Then the process is suspended until a signal is caught or until a signal occurs that terminates the process. If a signal is caught and if the signal handler returns, then `sigsuspend` returns, and the signal mask of the process is set to its value before the call to `sigsuspend`.
- Note that there is no successful return from this function. If it returns to the caller, it always returns -1 with `errno` set to `EINTR`.
- Example (Figure 10.22, `signals/suspend1.c`) shows the correct way to protect a critical region of code from a specific signal.
- Another use of `sigsuspend` is to wait for a signal handler to set a global variable. In the program shown in Figure 10.23 (`signals/suspend2.c`), we catch both the interrupt signal and the quit signal, but want to wake up the main routine only when the quit signal is caught.
- Example (Figure 10.23, `signals/suspend2.c`).
- Example (Figure 10.24, `lib/tellwait.c`) is an another example of signals shows how signals can be used to synchronize a parent and child.

15 abort Function

- The `abort` function sends the `SIGABRT` signal to the caller and causes abnormal program termination.

```
#include <stdlib.h>
void abort(void);
```

- ISO C states that calling `abort` will deliver an unsuccessful termination notification to the host environment by calling `raise(SIGABRT)`.
- ISO C requires that if the signal is caught and the signal handler returns, `abort` still doesn't return to its caller.
- POSIX.1 also specifies that `abort` overrides the blocking or ignoring of the signal by the process.
- Example (Figure 10.25, `signals/abort.c`).

16 system Function

- In Section 8.13, we showed an implementation of the `system` function. That version, however, did not do any signal handling. POSIX.1 requires that `system` ignore `SIGINT` and `SIGQUIT` and block `SIGCHLD`.
- Example (Figure 10.26, `signals/systest2.c`) shows why we need to worry about signal handling.
- Next figure (Figure 10.27 in textbook) shows the arrangement of the processes when the editor is running.
- Example (Figure 10.28, `signals/system.c`) shows an implementation of the `system` function with the required signal handling.

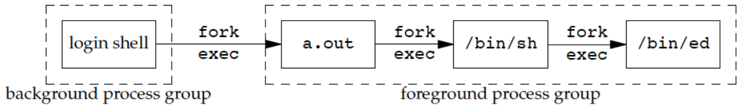

Figure 1: Foreground and background process groups for Figure 10.26

17 `sleep`, `nanosleep`, and `clock_nanosleep` Functions

- The `sleep` function

```
#include <unistd.h>
unsigned int sleep(unsigned int seconds);
```

  returns 0 or number of unslept seconds

- This function causes the calling process to be suspended until either
  1. The amount of wall clock time specified by *seconds* has elapsed.
  2. A signal is caught by the process and the signal handler returns.
- Figure 10.29 (`lib/sleep.c`) shows an implementation of the POSIX.1 `sleep` function.
- The `nanosleep` function is similar to the `sleep` function, but provides nanosecond-level granularity.

```
#include <time.h>
int nanosleep(const struct timespec *reqtp,
              struct timespec *remtp);
```

- If the system doesn't support nanosecond granularity, the requested time is rounded up.
- Because the `nanosleep` function doesn't involve the generation of any signals, we can use it without worrying about interactions with other functions.
- With the introduction of multiple system clocks, we need a way to suspend the calling thread using a delay time relative to a particular clock. The `clock_nanosleep` function provides us with this capability.

```
#include <time.h>
int clock_nanosleep(clockid_t clock_id, int flags,
                    const struct timespec *reqtp,
                    struct timespec *remtp);
```

18 `sigqueue` Function

- In Section 10.8 we said that most UNIX systems don't queue signals. With the real-time extensions to POSIX. 1, some systems began adding support for queueing signals. With SUSv4, the queued signal functionality has moved from the real-time extensions to the base specification.
- Generally a signal carries one bit of information: the signal itself. In addition to queueing signals, these extensions allow applications to pass more information along with the delivery (recall Section 10.14). This information is embedded in a `siginfo` structure. Along with system-provided information, applications can pass an integer or a pointer to a buffer containing more information to the signal handler.
- To use queued signals we have to do the following:

1. Specify the SA_SIGINFO flag when we install a signal handler using the sigaction function. If we don't specify this flag, the signal will be posted, but it is left up to the implementation whether the signal is queued.
2. Provide a signal handler in the sa_sigaction member of the sigaction structure instead of using the usual sa_handler field. Implementations might allow us to use the sa_handler field, but we won't be able to obtain the extra information sent with the sigqueue function.
3. Use the sigqueue function to send signals.

```
#include <signal.h>
int sigqueue(pid_t pid, int signo,
             const union sigval value)
```

- It returns 0 if OK, -1 on error.
- The sigqueue function is similar to the kill function, except that we can only direct signals to a single process with sigqueue, and we can use the *value* argument to transmit either an integer or a pointer value to the signal handler.
- Signals can't be queued infinitely. When this limit is reached, sigqueue can fail with errno set to EAGAIN.
- With the real-time signal enhancements, a separate set of signals was introduced for application use. These are the signal numbers between SIGRTMIN and SIGRTMAX, inclusive. Be aware that the default action for these signals is to terminate the process.
- Figure 10.30 in the textbook summarizes the way queued signals differ in behavior among the implementations covered in this text.

19 Job-Control Signals

- Of the signals shown in Figure 10.1, POSIX.1 considers six to be job-control signals:

  SIGCHLD Child process has stopped or terminated.

  SIGCONT Continue process, if stopped.

  SIGSTOP Stop signal (can't be caught or ignored).

  SIGTSTP Interactive stop signal.

  SIGTTIN Read from controlling terminal by background process group member.

  SIGTTOU Write to controlling terminal by a background process group member.
- The program in Figure 10.31 (signals/sigtstp.c) demonstrates the normal sequence of code used when a program handles job control.

20 Signal Names and Numbers

- In this section, we describe how to map between signal numbers and names.
- Some systems provide the array[1]:

```
extern char *sys_siglist[];
```

[1] Solaris 10 uses the name _sys_siglist instead.

The array index is the signal number, giving a pointer to the character string name of the signal.

- To print the character string corresponding to a signal number in a portable manner, we can use the psignal function.

```
#include <signal.h>
void psignal(int signo, const char *msg);
```

- The string *msg* is output to the standard error, followed by a colon and a space, and a description of the signal, then a newline. If *msg* is `NULL`, then only the description is written to the standard error.
- If you have a `siginfo` structure from an alternative `sigaction` signal handler, you can print the signal information with the `psiginfo` function.

```
#include <signal.h>
void psiginfo(const siginfo_t *info, const char *msg);
```

- If you only need the string description of the signal and don't necessarily want to write it to standard error, you can use the `strsignal` function.

```
#include <string.h>
char *strsignal(int signo);
```

- Solaris provides a couple of functions to map a signal number to a signal name, and vice versa.

```
#include <signal.h>
int sig2str(int signo, char *str);
int str2sig(const char *str, int *signop);
```

- Note that `sig2str` and `str2sig` depart from common practice and don't set `errno` when they fail.

The End of Chapter 10.