Topic 6

Signal Handling in UNIX

Objectives

- Understand the concept of signal and signal handling mechanism
- Be able to write signal handlers
- Be able to catch and ignore signals
- Be able to block and unblock signals
- Be aware of limitations of the traditional signal mechanism
- Understand and be able to use the new, reliable signal model and its programming interface

Readings

- This lecture notes
- Stevens & Rago: Ch 10
- Stallings: Table 2.6 on page 124 and Table 6.2 on page 316.

1. Why Signals?

Signals are used to model asynchronous events (such as hardware interrupts). Asynchronous events are events that are likely to occur, but it is difficult to know whether they will occur. Even if you are sure that they will occur, you do not know when they will occur.

Examples of asynchronous events:

(1) Pressing `CTRL-C` to terminate a program

(2) Executing an illegal instruction

(3) Dividing by 0

(4) Child process notifying the parent process that it has terminated

(5) Operating system kernel notifying a process that its alarm clock had expired

(6) Reference to an illegal memory address

(7) Modem connection hanging up

(8) Operating system kernel notifying a process that its window size has changed

2. How to process asynchronous events?

(1) Polling the Device

For example, you may examine (poll) the terminal keyboard periodically to see whether the user has pressed the `CRTL-C` key.

Disadvantages:

(a) It wastes a lot of CPU time

(b) It is still possible to miss an event (which may be disastrous)

(c) It may need to poll many devices to ensure no event is missed

(d) It is difficult to write and understand such kind of programs

(2) Using the Interrupt Mechanism

A process keeps doing what it is supposed to do as if asynchronous events will never occur. However, once an event does occur, the interrupt mechanism would stop the execution of the current process, and then execute a piece of code for handling the event. After the code completes its execution, the OS kernel resumes the execution of the interrupted process.

3. Signals

Signals model the asynchronous events. UNIX defines a set of 31 signals, each of which is represented by an integer number. Each signal represents one (or one type of) event(s). The following table lists some of the signals. Figure 10.1 of Stevens and Rago provides a full list of signals available in several important Unix-like operating systems.

Signal Symbolic Name	Signal Numeric Value	Event
SIGHUP	1	modem connection hangs up
SIGINT	2	interrupt the foreground processes (CTRL-C)
SIGQUIT	3	terminate the foreground processes (CTRL-\)
SIGILL	4	Illegal instruction
SIGFPE	8	floating point error
SIGKILL	9	Kill (cannot be caught or ignored)
SIGSEGV	11	Invalid memory reference
SIGPIPE	13	write to pipe that no one reads
SIGALRM	14	Alarm clock expires
SIGURG	16	urgent condition occurred
SIGTSTP	18	stop the foreground processes (CTRL-Z)
SIGCHLD	20	child process terminates
SIGUSR1	30	user-defined signal
SIGUSR2	31	user-defined signal

For the full list of signls, consult the header file `<signal.h>` which is file `/usr/include/signal.h`.

4. Sending Signals

- A user can send some signals to all processes in the foreground process group by typing the following key combinations:

CTRL-C	SIGINT
CTRL-\	SIGQUIT
CTRL-Z	*SIGTSTP*

- The UNIX kernel may send some signals to a process, e.g.,

```
SIGALRM
```


- A process may send a signal to another process, either in the program using the system call `kill`:


```
retval = kill (pid, signo)
```


or from the command line using the command `kill`:

```
% kill -SIGKILL 1001
% kill -9 1001
```

- A process may generate some signals during its execution (actually generated by the hardware)

SIGILL	-	executing an illegal instruction
SIGSEGV	-	illegal address reference

5. Handling Signals

When a signal arrives, the receiving process may choose to:

(a) ignore the signal (so nothing happens)
(However, a process cannot ignore `SIGKILL`)

(b) stop the execution of the process to perform the default operation associated with the signal

(c) catch the signal, i.e., stop the execution of the process to perform a predefined operation

In the old UNIX system, signals were handled through the use of the `signal` function. Under the old signal model, a signal is either processed or ignored. If the receiving process does not have enough time to process all incoming signals immediately, some of those signals could be lost (without the receiving process knowing about it). Therefore, the old signal model is not reliable.

With the new signal model, the receiving process can block a signal if it does not want to process the signal immediately for whatever reason. The blocked signal is not lost, the process can unblock it later and then process it. The new signal model is reliable but has a more complicated programming interface.

In this topic, we will use the old `signal` function first, and discuss the new programming interface for reliable signal model later.

However, when writing a new program, you should use only the reliable signal interface (eg, `sigaction`), not the `signal` function. The function `signal` is now implemented with the reliable signal model (so it is reliable), however, it behaves differently on different Unix operating systems.

6. Function Pointers

Before introducing the signal function, let's review a couple of C language features. In C, a function is not a variable. However, we can declare a pointer variable that points to a particular type of functions. We can then invoke the function using that pointer variable instead of the function name.

Example 1: A void function that takes an integer parameter

```
void f1(int i)
{
   printf("This is function f1, i = %d\n", i);
}

void f2(int i)
{
   printf("This is function f2, i = %d\n", i);
}
```


Note: both `f1` and `f2` are functions, not variables.

Example 2: A function pointer

The variable `f` in the following declaration is a pointer variable that points to a void function that takes an integer argument, which is similar to the functions such as `f1` and `f2` defined above.

```
void (*f) (int i);
```

Example 3: Compare the following three declarations:

```
(1) int ff (int n);
(2) int *ff (int n);
(3) int (*ff) (int n);
```

In the above three declarations, the first `ff` is an integer function that takes an integer parameter. The second `ff` is a function that takes an integer parameter and returns a pointer to an integer. The third `ff` is a pointer to a function that takes an integer parameter and returns an integer.

Example 4: Using a function pointer

Since the variable `f` declared in Example 2 is a pointer variable, we may assign the address of a *compatible* function to it. In C, the function name itself, if used without parentheses, represents the starting address to that function. This is like an array variable whose name represents the starting address of the array. Therefore, we can do:


```
f = f1;
f(100);          // same as f1(100);
f = f2;
f(200);          // same as f2(200);
```


Compare this to an array and a pointer:

```
char line[] = "hello";    // array
char *s;                  // pointer

s = line;         // assign the start address
                  // of the array to pointer
                  // variable s

printf ("s = %s \n", s);  // print the string
                          // "hello"
```

7. Setting Signal Handler

(1) Signal Function (old)

```
#include <signal.h>

void (*signal( int signo, void (*handler)(int) ) )(int);
```

Install the new signal handler `handler` for signal `signo` and return the previous signal handler for signal `signo`.

It is not easy to read the above function prototype. However, if you focus on the red and green part from the above declaration, it would be easy to see

- that `signal` is just a function that takes an integer `signo` and a function pointer `handler` (to a void function that takes an integer parameter).
- This function (`signal`) returns a pointer to a void function that takes an integer parameter.

An alternative way to look at this function prototype is by redefine the `signal` with a new type (a void function taking an integer parameter):

```
typedef void sigfunc(int);
```

to define `signal` function:

```
sigfunc * signal(int signo, sigfunc * handler);
```

That is:

- `signal` is a function. It takes two parameters `signo` and `handler`.
- It returns a pointer to a void function that takes one integer parameter;

(2) Set the Default Signal Handler

```
signal (sig, SIG_DFL);
```

The system defined constant `SIGN_DFL` represents the default signal handler. The default operation for most signals is to terminate the receiving process.

Example 5: Kill a foreground process using `CTRL-C`

```
% top
```

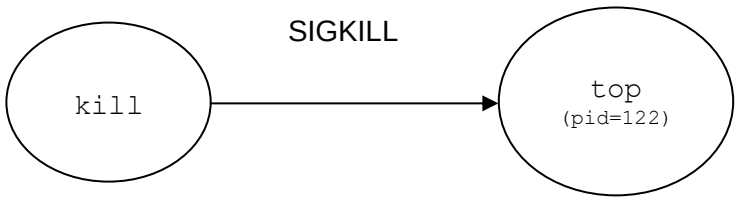
If you press "CTRL-C" on the terminal, the kernel would send the signal `SIGINT` to the foreground process `top` which would then take the default action -- terminate the itself.

Example 6: Send a signal to a background process


```
%  top  &
```



```
%  kill  -SIGKILL    122
```



(3) Ignore a Signals

```
signal (sig, SIG_IGN);
```

`SIG_IGN` is a system defined constant.

Example 7: Ignore a signal

```
#include <signal.h>

int main()
{
   signal (SIGINT, SIG_IGN);

   for ( ; ; ). // infinite loop
   printf("You cannot interrupt me!\n");
}
```


- type `CTRL-C` (`SIGINT`) will not stop this process
- type `CTRL-\` (`SIGQUIT`) will do.

To restore the default signal handler, use:

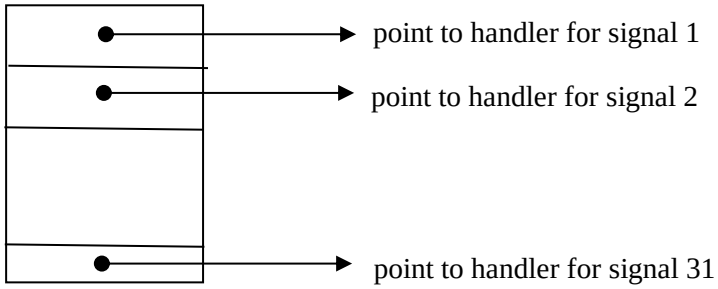
```
signal (sig, SIG_DFL);
```

Example 8: Ignore a signal and then restore the signal handler

```
#include <signal.h>

int main()
{
   signal(SIGINT, SIG_IGN);

   for (int i = 0; i < 20; ++i) {
      printf("You cannot stop me now!\n);
      sleep (1);
   }

   signal(SIGINT, SIG_DFL);

   for (  ;  ; ) {  // infinite loop
      printf("You can stop me now!\n");
      sleep (1);
   }
}
```

(4) Catch a Signal

Signal Table: for each process, the kernel keeps a signal table (in the kernel space, just like the Per Process Table of Open Files). The table keeps the address to the current signal handler for each signal.



When a signal arrives, the normal execution of the receiving process is stopped, and the kernel retrieves the address of the signal handler from the Signal Table and then calls the signal handler for that signal. The signal handler is just a void function that takes one integer parameter. The kernel passes the signal number as the integer argument to the signal handler function. Once the signal handler returns, the kernel resumes the execution of the interrupted process.

The user program may change the signal handler to a user-defined function using the signal function:

```
void (*oldfunc)(int);  // function pointer
void newfunc(int) { . . . };
oldfunc = signal (sig, newfunc);
```

Three special function pointers are defined in the Unix system, which have predefined meanings:

```
SIG_ERR:   (void (*) ())   -1   error
SIG_DFL:   (void (*) ())   0    default operation
SIG_IGN:   (void (*) ())   1    ignore this signal
```

Example 9: Catch `SIGINT` signals with `signal` function

```
#include <signal.h>

void catch(int signo)
{
    printf("Signal %d is caught.\n", signo);
}

int main()
{
   signal(SIGINT, catch);

   for (int i=0; i < 100; ++i)
   {
      printf("Sleep call %d\n", i);
      sleep(1);
   }
   printf("Exiting.\n");
   exit(0);
}
```


- Pressing `CTL-C` would not terminate the program, because signal `SIGINT` would be caught by the program.
- Similarly, using `"kill -SIGINT pid"` would not kill the program.

8. Alarm and Pause

The following functions/system calls all have something to do with time. Here is a comparison:

wait: `pid = wait(&status)`
Suspend the calling process until one of its child processes terminates

sleep: `sleep(n)`
Suspend the calling process for a specified period of time (*n* seconds)

alarm: `remain = alarm(n)`
Set the calling process' alarm clock to *n* seconds and return immediately. When the timer expires, the kernel will send the signal `SIGALRM` to the process.

pause: `pause()`
Suspend the calling process until a signal is received and the signal handler is executed and returned.

Example 10: Use the alarm clock

```
/*    tml - tell me later program (adapted from Haviland)
 *    usage example: tml  30  it is time to go home
 */
#include <unistd.h>
#include <stdio.h>
#include <signal.h>

int alarm_expired = 0; // set it to 1 once the alarm has expired
void setflag (int signo) { alarm_expired = 1;} // signal handler

int main(int argc, char *argv[])
{
    int nsecs, j;
    pid_t pid;

    if (argc <=2)
        fprintf(stderr, "Usage: %s #minutes message\n", argv[0]),
exit(1);
    if ((nsecs = atoi(argv[1])*60) <= 0)
        fprintf(stderr, "%s: invalid time\n", argv[0]), exit(2);

    pid = fork();
    if (pid < 0)
        perror(" can't create process"), exit(3);
    else if (pid>0) // parent to quit, print child pid
        printf("%s: process id=%d\n", argv[0], pid), exit(0);

    // now only the child left
    signal(SIGALRM, setflag);   //install signal handler
    alarm(nsecs);     // turn on alarm clock
    pause();      // pause until signal

    // now a has signal arrived and its signal handler executed
    if (alarm_expired) {  // check whether this signal is SIGALRM
        printf("\007\007\007");  // ring bells
        for (i=2; i<agrc; ++i)   // send the message to terminal
            printf("%s ", argv[i]);
        printf("\n");
    }
    exit(0);
}
```

9. Unreliable Signals

There are some problems with the semantics of the traditional signal model: when a signal is caught by a process, should the signal handler for that signal be reset to the default handler?

SVR4 and 4.3BSD adopt different approaches.

(1) SVR4

When a signal is caught, the handler for the signal is reset to the default handler.

To avoid the default handler being reset, we can re-install the signal handler:


```
void catch(int signo)
{
    signal(signo, catch);
    printf("Signal %d caught\n", signo);
}
```


However, it is still possible that a signal arrives during a small window of time after the signal was caught but before the signal handler is re-installed. When this occurs, the process would terminate (performs the default operation).

(2) 4.3 BSD

The signal handler will remain unchanged when a signal is caught. But this definition would cause problems if the signal handler calls a function that is not reentrant-safe, such as the example below:

```
char name[MAX_SIZE];
char number[MAX_SIZE]

void getStudent()
{
    printf("Type student name:");
    fgets (name, MAX_SIZE, stdin);
    printf("Type student No:");
    fgets(number, MAX_SIZE, stdin);
}
```

If a new signal arrives while the handler is executing the statement:

```
printf("Type your student No:");
```


then the handler would be called again and the previous name will be overwritten.

Therefore, whatever approach we choose, there is always a problem. The root of the problem is that, in the traditional signal model, we must either

(1) immediately perform a predefined operation, or

(2) completely ignore the signal

In other words, when a signal arrives, the process must handle it immediately without delay. There is no provision for us to delay the handling of a signal until a later stage.

10. Reliable Signals

To rectify this problem, the UNIX system introduced *reliable signals* – signals that would not get lost. With the reliable signal semantics, the life-time of a signal is divided into the following three stages:

(1) A signal is *generated* or sent to a process when the event that causes the signal occurs (e.g., pressing `CTRL-C`, dividing by 0, executing illegal instructions).

(2) The signal is *delivered* when the action associated with the signal is taken.

(3) The signal is said to be ***pending*** between generation and delivery.

Usually, a signal would be delivered immediately, but a process has the option to block some signals. If a signal that is blocked is generated for a process, and if the action for the signal is either the default action or to catch the signal (i.e., if it is not to be ignored), then the signal remains *pending* for the process until either:

- The process unblocks the signal or
- The process changes the action to ignore the signal.

11. Reliable Signal Interface (POSIX.1 compliant)

The programming interface for the reliable signals centres on the function `sigaction` and the type `struct sigaction`. These functions and types are defined in POSIX.1. Unix systems also provide the function `signal` **(**`signal` function is defined in ANSI C)**,** which is implemented using `sigaction` in most systems. However, its behaviour may vary in different operating systems. For the sake of portability, we should use `sigaction function` rather than `signal` function to install signal handlers.

Signal Sets

We can define a set of signals using the type `sigset_t` and the following functions:


```
#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 (sigset_t *set, int signo)
```

The functions `sigemptyset` and `sigfillset` are used to set a signal set to empty or to full set of signals. The functions `sigaddset` and `sigdelset` are used to add or delete an individual signal to or from a given signal set. These functions return 0 if successful or -1 on error. The function `sigismember` is used to test whether the given signal is in the specified signal set. This function returns 1 if the signal is in the set or 0 if it is not.

Example 11: Create a sigset

The following code creates a signal set `sigs` with two signals: `SIGINT` and `SIGQUIT`.

```
#include <signal.h>
sigset_t sigs;
if (sigemptyset(&sigs)==0) {
   sigaddset(&sigs, SIGINT);
   sigaddset(&sigs, SIGQUIT);
}
```

Setting the Signal Handler

We use the following function to change the signal handler:


```
int sigaction( int signo,
               const struct sigaction *act,
               struct sigaction *oldact);
```


act points to the structure containing the information of the new handler for the signal **signo**. If **oldact** is not `NULL`, the structure containing the existing signal handler is copied to **oldact**.

The `sigaction` structure is defined as

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

where one of the two function pointers, either `sa_handler` or `sa_sigaction`, points to the signal handler function. If `sa_flags` does not contain `SA_SIGINFO` flag then `sa_handler`

points to the new signal handler function while `sa_sigaction` is redundant. Otherwise, `sa_sigaction` points to the new signal handler with three arguments:

```
void func_name(int, siginfo_t *, void * );
```

When a signal is received, the signal handler pointed to by `sa_handler` or `sa_action` is called. During the execution of the signal handler, additional signals with the same signal number are automatically blocked until the execution is finished.

Other signals can also be blocked during the execution of the signal handler. This set of signals is defined in `sa_mask`.

`sa_flags` modifies the behaviors of the signal handling process. For example if `sa_flags=SA_RESETHAND`, then the signal handler is reset to the default once the handler is called (as in SVR4). Note POSIX.1 only defines one flag: `SA_NOCLDSTOP` (if signo is `SIGCHLD`, do not receive notification when child processes stop, e.g., when a child process is stopped by `CTRL-Z`).

Example 12: Catch `SIGINT` signals with `signaction`

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

void catch(int signo)
{
    printf("Signal %d caught\n", signo);
    sleep (4);
    printf("Signal %d returns\n", signo);
}

int main()
{
    int i;
    struct sigaction act;

    act.sa_flags = 0;
    act.sa_handler = catch;
    sigfullset( &(act.sa_mask) ); //to block all

    if (sigaction(SIGINT, &act, NULL) != 0) {
        perror("sigaction"); exit(1);
    }

    for (i=0; i<10; ++i) {
        printf("sleep %d ....\n", i); sleep (10);
    }

    exit(0);
}
```


The test output:

```
sleep 0 ....
sleep 1 ....
sleep 2 ....
Signal 2 caught
Signal 2 returns
Signal 2 caught
Signal 2 returns
Quit (core dumped)
```

In the above test run, we had pressed `CTRL-C` three times within 2 seconds and followed by one `CTRL-\`.

The `CTRL-\` caused the process to terminate with a core dump (memory dump). This was because we did not catch `SIGQUIT`, hence once the process received a `SIGQUIT`, it performed the default operation associated with `SIGQUIT`, i.e., to terminate with a core dump.

From the above output, it seems that the signal handler `catch` was called only twice, even though `CTRL-C` was pressed three times in quick succession. What happened to the third `SIGINT`?

Actually, POSIX.1 does not require the queuing of the blocked signals. When the first `SIGINT` is being processed, the additional occurrences of `SIGINT` are blocked, but these additional `SIGINT`s are not queued, i.e., if two `SIGINT`s arrive while they are blocked, only one is kept. This is why only two `SIGINT`s were processed.

12. Signal Blocking

In some applications, it may be required that the execution of a critical section of an application not to be interrupted by certain types of incoming signals. This is achieved by blocking those signals before entering that *critical section* and then unblocked those signals after the critical section has been executed.

POSIX.1 provides the following function to block and unblock a set of signals:

```
#include <signal.h>
int sigprocmask(
        int how,
        const sigset_t *bset,
        sigset_t * oldset);
```


the parameter `how` is one of the following values:

`how=SIG_SETMASK`
: To block the set of signals specified in parameter `bset`.

`How=SIG_BLOCK`
: To add the set of signals in parameter `bset` to the set of signals to be blocked, and block the resultant set of signals.

`How=SIG_UNBLOCK`
: To unblock the set of signals specified in parameter `bset`.

Example 13: Signal blocking

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

int main()
{
    int i;
    sigset_t sigs, sigs2;

    sigemptyset(&sigs);
    sigaddset(&sigs, SIGINT);
    sigaddset(&sigs, SIGQUIT);

    sigprocmask(SIG_SETMASK, &sigs, NULL);

    /* simulating a critical section */
    printf("let me finish before you kill me ...\n");
    for (i=0; i<10; ++i) {
        printf("working ... \n");
        sleep (1);
    }
    printf("The work is done, now you can ...\n");
    /* end of the critical section */

    sigprocmask(SIG_UNBLOCK, &sigs, NULL);

    exit(0);
}
```

The test output:

```
let me finish before you kill me ...
working ....
working ....
working ....
working ....
working ....
working ....
working ....
working ....
working ....
working ....
The work is done, now you can ...
Quit (core dumped)
```


In the above test run, we had pressed `CTRL-\` while the process was "working". The process was not killed until it had finished its "work". This was because `SIGQUIT` was blocked while the process was "working", and unblocked after the "work" was done.

The following function is used to find out which signals are currently pending:


```
#include <signal.h>

int sigpending(sigset_t *set);
```

On return the `set` contains the set of signals that are currently pending.

Example 14: Signal pending

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

int main()
{
    int i;
    sigset_t sigs, sigs2;

    sigemptyset(&sigs);
    sigaddset(&sigs, SIGINT);
    sigaddset(&sigs, SIGQUIT);

    sigprocmask(SIG_SETMASK, &sigs, NULL);

    /* critical section */
    printf("let me finish before you kill me ...\n");
    for (i=0; i<10; ++i) {
        printf("working ... \r");
        sleep (1);
        sigpending (&sigs2);
        if (sigismember(&sigs2, SIGINT))
            printf("SIGINT is pending\n");
        if (sigismember(&sigs2, SIGQUIT))
            printf("SIGQUIT is pending\n");
    }
    printf("The work is done, now you can ...\n");
    /* end of the critical section */

    sigprocmask(SIG_UNBLOCK, &sigs, NULL);


    exit(0);
}
```

The test output:

```
let me finish before you kill me ...
working ....
working ....
working ....
SIGQUIT is pending
working ....
SIGQUIT is pending
working ....
SIGQUIT is pending
working ....
SIGQUIT is pending
working ....
SIGQUIT is pending
working ....
SIGQUIT is pending
working ....
SIGQUIT is pending
working ....
SIGQUIT is pending
The work is done, now you can ...
Quit (core dumped)
```