

Signals and Signal Handling - Part 1

What Is Signals?

- A primitive way of doing IPC and most widely known UNIX facility.
 - Be used to inform processes of *asynchronous* events.
 - Posted by one process and received by another or the same process.
 - An asynchronous event either terminates a process or is simply being ignored.
 - Signal handling
 - Default (SIG_DFL)
 - Ignored (SIG_IGN)
 - User-defined

Signal Generation

- A signal is generated when (not a complete list):
 - A hardware exception occurs.
 - Interrupt or quit from control terminal.
 - An alarm timer expires.
 - A call to kill().
 - Termination of a child process.

Signal Usage



- Signals can be used:
 - Intraprocess
 - With the same user ID
 - Interprocesses
 - Between kernel to any process.

Signal States



- A signal is said to be:
 - *Generated* when the event that causes the signal occurs.
 - *Delivered* when the action for a signal is taken.
 - *Pending* during the time between the generation of the signal and its delivery.
 - *Blocked* if unable to deliver due to a signal mask bit being set for the signal.

Signal Disposition

- Default action (SIG_DFL)
 - Termination in general.
- Ignored (SIG_IGN)
 - Never posted to the process.
- User-defined action
 - Needs a user-defined *signal handler*, or *signal-catching function*.
 - Most signals can be caught, or ignored except SIGKILL and SIGSTOP.

Linux Signals (1)

SIGHUP	1 /* Hangup (POSIX). terminate w/ core */
SIGINT	2 /* Interrupt (ANSI). terminate */
SIGQUIT	3 /* Quit (POSIX). terminate w/ core */
SIGILL	4 /* Illegal instruction (ANSI). terminate w/ core */
SIGTRAP	5 /* Trace trap (POSIX). terminate w/ core */
SIGABRT	6 /* Abort (ANSI). terminate w/ core */
SIGIOT	6 /* IOT trap (4.2 BSD). terminate w/ core */
SIGBUS	7 /* BUS error (4.2 BSD). terminate w/ core */
SIGFPE	8 /* Floating-point exception (ANSI). terminate w/ core */
SIGKILL	9 /* Kill, unmaskable (POSIX). terminate */
SIGUSR1	10 /* User-defined signal 1 (POSIX). terminate */

Linux Signals (2)

SIGSEGV	11 /* Segmentation violation(ANSI). terminate w/ core */
SIGUSR2	12 /* User-defined signal 2 (POSIX). terminate */
SIGPIPE	13 /* Broken pipe (POSIX). terminate */
SIGALRM	14 /* Alarm clock (POSIX). terminate */
SIGTERM	15 /* Termination (ANSI). terminate */
SIGSTKFLT	16 /* Stack fault. terminate w/ core */
SIGCLD	SIGCHLD /* Same as SIGCHLD (System V). */
SIGCHLD	17 /* Child status has changed(POSIX). ignore */
SIGCONT	18 /* Continue (POSIX). continue/ignore */
SIGSTOP	19 /* Stop, unmaskable (POSIX). stop process */
SIGTSTP	20 /* Keyboard stop (POSIX). stop process */
SIGTTIN	21 /* Background read from tty (POSIX). stop process */
SIGTTOU	22 /* Background write to tty (POSIX). stop process */

Linux Signals (3)

SIGURG	23	/* Urgent condition on socket (BSD). ignore */
SIGXCPU	24	/* CPU limit exceeded (BSD). terminate w/ core */
SIGXFSZ	25	/* File size limit exceeded (BSD). terminate w/ core */
SIGVTALRM	26	/* Virtual alarm clock (BSD). terminate */
SIGPROF	27	/* Profiling alarm clock (BSD). terminate */
SIGWINCH	28	/* Window size change (BSD,Sun). ignore */
SIGPOLL	SIGIO	/* Pollable event occurred (System V). terminate */
SIGIO	29	/* I/O now possible (BSD). terminate/ignore */
SIGPWR	30	/* Power failure restart (System V). ignore */
SIGUNUSED	31	

Signal Sets

- Signal sets are one of the main parameters passed to system calls that deal with signals
- A list of signals you want to do something with
- To manipulate signal sets, a new data type known as `sigset_t` with the following five predefined functions is specified in POSIX.1:
 - `sigemptyset()`
 - `sigfillset()`
 - `sigaddset()`
 - `sigdelset()`
 - `sigismember()`

Signal Set (1)

- `#include <signal.h>`
`int sigemptyset(sigset_t *set);`
 - Initializes the signal set given by `set` to empty, with all signals excluded from the set.
 - Return 0 on success and -1 on error.
- `int sigfillset(sigset_t *set);`
 - Initializes `set` to full, including all signals.
 - Return 0 on success and -1 on error.

Signal Set (2)

- `int sigaddset(sigset_t *set, int signum);`
 - Add signal *signum* from *set*.
 - Return 0 on success and -1 on error.
- `int sigdelset(sigset_t *set, int signum);`
 - Delete signal *signum* from *set*.
 - Return 0 on success and -1 on error.
- `int sigismember(const sigset_t *set, int signum);`
 - Tests whether *signum* is a member of *set*.
 - Returns 1 if *signum* is a member of *set*, 0 if *signum* is not a member, and -1 on error.

Example #1: Signal Sets

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

int main(){
    sigset_t mask1, mask2;
    //..
    /* create empty set */
    sigemptyset(&mask1);

    /* add signals */
    sigaddset(&mask1, SIGINT);
    sigaddset(&mask1, SIGQUIT);
    //..
    /* create full set */
    sigfillset(&mask2);

    /* remove signal */
    sigdelset(&mask2, SIGCHLD);
    //..
}
```

sigaction (1)

- `#include <signal.h>`

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

- Change the action taken by a process on receipt of a specific signal.
- Return 0 on success and -1 on error.
- *signum*
 - Specifies the signal and can be any valid signal except SIGKILL and SIGSTOP.
- If *act* is non-NULL, the new action for signal *signum* is installed from *act*.
- If *oldact* is non-null, the previous action is saved in *oldact*.

sigaction (2)

– sigaction structure

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

– sa_handler

- Specifies the action to be associated with *signum* and may be SIG_DFL for the default action.
- SIG_IGN to ignore this signal, or a pointer to a signal handling function.

sigaction (3)

- `sa_sigaction(int, siginfo_t *, void *)`
 - If `sa_flags` is set to `SA_SIGINFO`, extra information will be passed to the signal handler. In this case, `sa_sigaction()` is used
- `sa_mask`
 - Gives a mask of signals which should be blocked during execution of the signal handler.
- `sa_flags`
 - Specifies a set of flags which modify the behavior of the signal handling process.
 - It is formed by the bitwise OR of zero or more of the following:
 - `SA_NOCLDSTOP`
 - » If *signum* is `SIGCHLD`, do not receive notification when child processes stop.
 - `SA_RESETHAND`
 - » Restore the signal action to the default state once the signal handler has been called.

Example #2: catching SIGINT (1)

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

void catchint (int signo) {
    printf ("\nCATCHINT: signo=%d\n", signo);
    printf("CATCHINT: returning\n\n");
    return;
}

int main() {
    static struct sigaction act;
    act.sa_handler = catchint;

    sigfillset(&(act.sa_mask));
    sigaction(SIGINT, &act, NULL);

    printf ("sleep call #1\n");
    sleep (1);
}
```

Example #2: catching SIGINT (2)

```
printf ("sleep call #2\n");  
sleep (1);  
  
printf ("sleep call #3\n");  
sleep (1);  
  
printf ("sleep call #4\n");  
sleep (1);  
  
printf ("Exiting\n");  
return 0;  
}
```

```
➤ ./main  
sleep call #1  
^C  
CATCHINT: signo=2  
CATCHINT: returning  
  
sleep call #2  
sleep call #3  
sleep call #4  
^C  
CATCHINT: signo=2  
CATCHINT: returning  
  
Exiting
```

Example #3: ignoring SIGINT

- Just replace the following line in the example #2 program

- `act.sa_handler = catchint;`

With:

- `act.sa_handler = SIG_IGN;`

And then call `sigaction(SIGINT, &act, NULL)`

Example #4: restoring a previous action

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

```
static struct sigaction act, oact;
```

```
/* save the old action for SIGTERM */  
sigaction(SIGTERM, NULL, &oact);
```

```
/* set new action for SIGTERM */  
sigaction(SIGTERM, &act, NULL);
```

```
/* do the work here... */
```

```
/* now restore the old action */  
sigaction(SIGTERM, &oact, NULL);
```

Example #5: graceful exit

- Suppose a program uses a temporary workfile

```
/* exit from program gracefully */  
#include <stdio.h>  
#include <stdlib.h>  
#include <unistd.h>  
#include <signal.h>  
void g_exit(int s) {  
    unlink("tempfile");  
    fprintf(stderr, "Interrupted – exiting\n");  
    exit(1);  
}  
int main() {  
    // ...  
    /* in somewhere */  
    static struct sigaction act;  
    act.sa_handler = g_exit;  
    sigaction(SIGINT, &act, NULL);  
    // ...  
    return 0;  
}
```

Signal Handler

- `void (*signal(int signo, void (*handler)(int)))(int)`
 - Signal handler can be set by user process.
- `signal()` is said to be unreliable.
 - Signals can get lost.
- Further superseded by `sigaction()` in the latest implementations of various versions of UNIX systems.

signal (1)

- `#include <signal.h>`

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

- Installs a new signal handler for the signal with number *signum*.
- The signal handler is set to *handler* which may be a user specified function, or one of the following:
 - SIG_IGN: Ignore the signal.
 - SIG_DFL: Reset the signal to its default behavior.

signal (2)



- The integer argument that is handed over to the signal handler routine is the *signal number*.
- It is possible to use one signal handler for several signals.
- Return value
 - The previous value (address) of the signal handler, or SIG_ERR on error.

Example #6: signal (1)

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <signal.h>
#include <sys/types.h>

static void sig_usr(int signo) {
    if (signo == SIGUSR1)
        printf("received SIGUSR1\n");
    else if (signo == SIGUSR2)
        printf("received SIGUSR2\n");
    else {
        fprintf(stderr, "received signal %d\n", signo);
        fflush(stderr);
        abort();
    }
    return;
}
```

Example #6: signal (2)

```
int main(void) {  
    if (signal(SIGUSR1, sig_usr) == SIG_ERR) {  
        perror("can't catch SIGUER1");  
        exit(1);  
    }  
    if (signal(SIGUSR2, sig_usr) == SIG_ERR) {  
        perror("can't catch SIGUER2");  
        exit(1);  
    }  
  
    for(;;)  
        pause();  
}
```

```
❖ ./main &  
[1] 327  
❖ kill -USR1 327  
❖ received SIGUSR1  
kill -USR2 327  
❖ received SIGUSR2  
kill 327
```