Rocky Enterprise Linux 9.2 Manual Pages on command 'signalfd.2'

C:\>man signalfd.2

SIGNALFD(2) Linux Programmer's Manual SIGNALFD(2)

NAME

signalfd - create a file descriptor for accepting signals

SYNOPSIS

#include <sys/signalfd.h>

int signalfd(int fd, const sigset_t *mask, int flags);

DESCRIPTION

signalfd() creates a file descriptor that can be used to accept signals targeted at the caller. This provides an alternative to the use of a signal handler or sig? waitinfo(2), and has the advantage that the file descriptor may be monitored by se? lect(2), poll(2), and epoll(7).

The mask argument specifies the set of signals that the caller wishes to accept via the file descriptor. This argument is a signal set whose contents can be initial? ized using the macros described in sigsetops(3). Normally, the set of signals to be received via the file descriptor should be blocked using sigprocmask(2), to pre? vent the signals being handled according to their default dispositions. It is not possible to receive SIGKILL or SIGSTOP signals via a signalfd file descriptor; these signals are silently ignored if specified in mask.

If the fd argument is -1, then the call creates a new file descriptor and asso? ciates the signal set specified in mask with that file descriptor. If fd is not -1, then it must specify a valid existing signalfd file descriptor, and mask is used to replace the signal set associated with that file descriptor.

Starting with Linux 2.6.27, the following values may be bitwise ORed in flags to change the behavior of signalfd():

SFD_NONBLOCK Set the O_NONBLOCK file status flag on the open file description (see open(2)) referred to by the new file descriptor. Using this flag saves extra calls to fcntl(2) to achieve the same result.

SFD_CLOEXEC Set the close-on-exec (FD_CLOEXEC) flag on the new file descriptor. See the description of the O_CLOEXEC flag in open(2) for reasons why this may be useful.

In Linux up to version 2.6.26, the flags argument is unused, and must be specified as zero.

signalfd() returns a file descriptor that supports the following operations:

read(2)

If one or more of the signals specified in mask is pending for the process, then the buffer supplied to read(2) is used to return one or more sig? nalfd_siginfo structures (see below) that describe the signals. The read(2) returns information for as many signals as are pending and will fit in the supplied buffer. The buffer must be at least sizeof(struct signalfd_sig? info) bytes. The return value of the read(2) is the total number of bytes read.

As a consequence of the read(2), the signals are consumed, so that they are no longer pending for the process (i.e., will not be caught by signal han? dlers, and cannot be accepted using sigwaitinfo(2)).

If none of the signals in mask is pending for the process, then the read(2) either blocks until one of the signals in mask is generated for the process, or fails with the error EAGAIN if the file descriptor has been made non? blocking.

poll(2), select(2) (and similar)

The file descriptor is readable (the select(2) readfds argument; the poll(2) POLLIN flag) if one or more of the signals in mask is pending for the process.

The signalfd file descriptor also supports the other file-descriptor multi? plexing APIs: pselect(2), ppoll(2), and epoll(7).

close(2)

When the file descriptor is no longer required it should be closed. When all file descriptors associated with the same signalfd object have been closed, the resources for object are freed by the kernel.

The signalfd_siginfo structure

The format of the signalfd_siginfo structure(s) returned by read(2)s from a sig? nalfd file descriptor is as follows:

```
struct signalfd_siginfo {
    uint32_t ssi_signo;    /* Signal number */
    int32_t  ssi_errno;    /* Error number (unused) */
    int32_t  ssi_code;     /* Signal code */
    uint32_t ssi_pid;      /* PID of sender */
    uint32_t ssi_uid;      /* Real UID of sender */
    int32_t  ssi_fd;       /* File descriptor (SIGIO) */
    uint32_t ssi_tid;      /* Kernel timer ID (POSIX timers)
    uint32_t ssi_band;     /* Band event (SIGIO) */
    uint32_t ssi_overrun;  /* POSIX timer overrun count */
    uint32_t ssi_trapno;   /* Trap number that caused signal */
    int32_t  ssi_status;   /* Exit status or signal (SIGCHLD) */
    int32_t  ssi_int;      /* Integer sent by sigqueue(3) */
    uint64_t ssi_ptr;      /* Pointer sent by sigqueue(3) */
    uint64_t ssi_utime;    /* User CPU time consumed (SIGCHLD) */
    uint64_t ssi_stime;    /* System CPU time consumed
                              (SIGCHLD) */
    uint64_t ssi_addr;     /* Address that generated signal
                              (for hardware-generated signals) */
    uint16_t ssi_addr_lsb; /* Least significant bit of address
                              (SIGBUS; since Linux 2.6.37)
    uint8_t  pad[X];       /* Pad size to 128 bytes (allow for
                              additional fields in the future) */
};
```

Each of the fields in this structure is analogous to the similarly named field in the siginfo_t structure. The siginfo_t structure is described in sigaction(2).

Not all fields in the returned signalfd_siginfo structure will be valid for a spe?

cific signal; the set of valid fields can be determined from the value returned in the ssi_code field. This field is the analog of the siginfo_t si_code field; see sigaction(2) for details.

fork(2) semantics

After a fork(2), the child inherits a copy of the signalfd file descriptor. A read(2) from the file descriptor in the child will return information about signals queued to the child.

Semantics of file descriptor passing

As with other file descriptors, signalfd file descriptors can be passed to another process via a UNIX domain socket (see unix(7)). In the receiving process, a read(2) from the received file descriptor will return information about signals queued to that process.

execve(2) semantics

Just like any other file descriptor, a signalfd file descriptor remains open across an execve(2), unless it has been marked for close-on-exec (see fcntl(2)). Any sig? nals that were available for reading before the execve(2) remain available to the newly loaded program. (This is analogous to traditional signal semantics, where a blocked signal that is pending remains pending across an execve(2).)

Thread semantics

The semantics of signalfd file descriptors in a multithreaded program mirror the standard semantics for signals. In other words, when a thread reads from a sig? nalfd file descriptor, it will read the signals that are directed to the thread it? self and the signals that are directed to the process (i.e., the entire thread group). (A thread will not be able to read signals that are directed to other threads in the process.)

epoll(7) semantics

If a process adds (via epoll_ctl(2)) a signalfd file descriptor to an epoll(7) in? stance, then epoll_wait(2) returns events only for signals sent to that process. In particular, if the process then uses fork() to create a child process, then the child will be able to read(2) signals that are sent to it using the signalfd file descriptor, but epoll_wait(2) will not indicate that the signalfd file descriptor is ready. In this scenario, a possible workaround is that after the fork(2), the child process can close the signalfd file descriptor that it inherited from the

parent process and then create another signalfd file descriptor and add it to the epoll instance. Alternatively, the parent and the child could delay creating their (separate) signalfd file descriptors and adding them to the epoll instance until after the call to fork(2).

RETURN VALUE

On success, signalfd() returns a signalfd file descriptor; this is either a new file descriptor (if fd was -1), or fd if fd was a valid signalfd file descriptor.

On error, -1 is returned and errno is set to indicate the error.

ERRORS

EBADF The fd file descriptor is not a valid file descriptor.

EINVAL fd is not a valid signalfd file descriptor.

EINVAL flags is invalid; or, in Linux 2.6.26 or earlier, flags is nonzero.

EMFILE The per-process limit on the number of open file descriptors has been reached.

ENFILE The system-wide limit on the total number of open files has been reached.

ENODEV Could not mount (internal) anonymous inode device.

ENOMEM There was insufficient memory to create a new signalfd file descriptor.

VERSIONS

signalfd() is available on Linux since kernel 2.6.22. Working support is provided in glibc since version 2.8. The signalfd4() system call (see NOTES) is available on Linux since kernel 2.6.27.

CONFORMING TO

signalfd() and signalfd4() are Linux-specific.

NOTES

A process can create multiple signalfd file descriptors. This makes it possible to accept different signals on different file descriptors. (This may be useful if monitoring the file descriptors using select(2), poll(2), or epoll(7): the arrival of different signals will make different file descriptors ready.) If a signal ap? pears in the mask of more than one of the file descriptors, then occurrences of that signal can be read (once) from any one of the file descriptors.

Attempts to include SIGKILL and SIGSTOP in mask are silently ignored.

The signal mask employed by a signalfd file descriptor can be viewed via the entry for the corresponding file descriptor in the process's /proc/[pid]/fdinfo direc?

tory. See proc(5) for further details.

Limitations

The signalfd mechanism can't be used to receive signals that are synchronously gen? erated, such as the SIGSEGV signal that results from accessing an invalid memory address or the SIGFPE signal that results from an arithmetic error. Such signals can be caught only via signal handler.

As described above, in normal usage one blocks the signals that will be accepted via signalfd(). If spawning a child process to execute a helper program (that does not need the signalfd file descriptor), then, after the call to fork(2), you will normally want to unblock those signals before calling execve(2), so that the helper program can see any signals that it expects to see. Be aware, however, that this won't be possible in the case of a helper program spawned behind the scenes by any library function that the program may call. In such cases, one must fall back to using a traditional signal handler that writes to a file descriptor monitored by select(2), poll(2), or epoll(7).

C library/kernel differences

The underlying Linux system call requires an additional argument, size_t sizemask, which specifies the size of the mask argument. The glibc signalfd() wrapper func? tion does not include this argument, since it provides the required value for the underlying system call.

There are two underlying Linux system calls: signalfd() and the more recent sig? nalfd4(). The former system call does not implement a flags argument. The latter system call implements the flags values described above. Starting with glibc 2.9, the signalfd() wrapper function will use signalfd4() where it is available.

BUGS

In kernels before 2.6.25, the ssi_ptr and ssi_int fields are not filled in with the data accompanying a signal sent by sigqueue(3).

EXAMPLE

The program below accepts the signals SIGINT and SIGQUIT via a signalfd file de? scriptor. The program terminates after accepting a SIGQUIT signal. The following shell session demonstrates the use of the program:

```
$ ./signalfd_demo
^C              # Control-C generates SIGINT
```

```
        Got SIGINT
        ^C
        Got SIGINT
        ^\                  # Control-\ generates SIGQUIT
        Got SIGQUIT
        $
```

Program source

```
    #include <sys/signalfd.h>
    #include <signal.h>
    #include <unistd.h>
    #include <stdlib.h>
    #include <stdio.h>
    #define handle_error(msg) \
        do { perror(msg); exit(EXIT_FAILURE); } while (0)
    int
    main(int argc, char *argv[])
    {
        sigset_t mask;
        int sfd;
        struct signalfd_siginfo fdsi;
        ssize_t s;
        sigemptyset(&mask);
        sigaddset(&mask, SIGINT);
        sigaddset(&mask, SIGQUIT);
        /* Block signals so that they aren't handled
           according to their default dispositions */
        if (sigprocmask(SIG_BLOCK, &mask, NULL) == -1)
            handle_error("sigprocmask");
        sfd = signalfd(-1, &mask, 0);
        if (sfd == -1)
            handle_error("signalfd");
        for (;;) {
            s = read(sfd, &fdsi, sizeof(struct signalfd_siginfo));
```

```
            if (s != sizeof(struct signalfd_siginfo))
                handle_error("read");
            if (fdsi.ssi_signo == SIGINT) {
                printf("Got SIGINT\n");
            } else if (fdsi.ssi_signo == SIGQUIT) {
                printf("Got SIGQUIT\n");
                exit(EXIT_SUCCESS);
            } else {
                printf("Read unexpected signal\n");
            }
        }
    }
```

SEE ALSO

eventfd(2), poll(2), read(2), select(2), sigaction(2), sigprocmask(2), sigwait? info(2), timerfd_create(2), sigsetops(3), sigwait(3), epoll(7), signal(7)

COLOPHON

This page is part of release 5.05 of the Linux man-pages project. A description of the project, information about reporting bugs, and the latest version of this page, can be found at https://www.kernel.org/doc/man-pages/.