*Full credit is given to the above companies including the OS that this PDF file was generated!*

## *Red Hat Enterprise Linux Release 9.2 Manual Pages on 'eventfd_write.3' command*

***$ man eventfd_write.3***

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

NAME

eventfd - create a file descriptor for event notification

SYNOPSIS

#include <sys/eventfd.h>

int eventfd(unsigned int initval, int flags);

DESCRIPTION

eventfd() creates an "eventfd object" that can be used as an event

wait/notify mechanism by user-space applications, and by the kernel to

notify user-space applications of events. The object contains an un?

signed 64-bit integer (uint64_t) counter that is maintained by the ker?

nel. This counter is initialized with the value specified in the argu?

ment initval.

As its return value, eventfd() returns a new file descriptor that can

be used to refer to the eventfd object.

The following values may be bitwise ORed in flags to change the behav?

ior of eventfd():

EFD_CLOEXEC (since Linux 2.6.27)

Set the close-on-exec (FD_CLOEXEC) flag on the new file descrip?

tor. See the description of the O_CLOEXEC flag in open(2) for

reasons why this may be useful.

EFD_NONBLOCK (since Linux 2.6.27)

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 re? sult.

EFD_SEMAPHORE (since Linux 2.6.30)

Provide semaphore-like semantics for reads from the new file de? scriptor. See below.

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

The following operations can be performed on the file descriptor re? turned by eventfd():

read(2)

Each successful read(2) returns an 8-byte integer. A read(2) fails with the error EINVAL if the size of the supplied buffer is less than 8 bytes.

The value returned by read(2) is in host byte order?that is, the native byte order for integers on the host machine.

The semantics of read(2) depend on whether the eventfd counter currently has a nonzero value and whether the EFD_SEMAPHORE flag was specified when creating the eventfd file descriptor:

* If EFD_SEMAPHORE was not specified and the eventfd counter has a nonzero value, then a read(2) returns 8 bytes contain? ing that value, and the counter's value is reset to zero.

* If EFD_SEMAPHORE was specified and the eventfd counter has a nonzero value, then a read(2) returns 8 bytes containing the value 1, and the counter's value is decremented by 1.

* If the eventfd counter is zero at the time of the call to read(2), then the call either blocks until the counter be? comes nonzero (at which time, the read(2) proceeds as de? scribed above) or fails with the error EAGAIN if the file de? scriptor has been made nonblocking.

write(2)

A write(2) call adds the 8-byte integer value supplied in its buffer to the counter. The maximum value that may be stored in the counter is the largest unsigned 64-bit value minus 1 (i.e., 0xfffffffffffffffe). If the addition would cause the counter's value to exceed the maximum, then the write(2) either blocks un? til a read(2) is performed on the file descriptor, or fails with the error EAGAIN if the file descriptor has been made nonblock? ing.

A write(2) fails with the error EINVAL if the size of the sup? plied buffer is less than 8 bytes, or if an attempt is made to write the value 0xffffffffffffffff.

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

The returned file descriptor supports poll(2) (and analogously epoll(7)) and select(2), as follows:

* The file descriptor is readable (the select(2) readfds argu? ment; the poll(2) POLLIN flag) if the counter has a value greater than 0.

* The file descriptor is writable (the select(2) writefds argu? ment; the poll(2) POLLOUT flag) if it is possible to write a value of at least "1" without blocking.

* If an overflow of the counter value was detected, then se? lect(2) indicates the file descriptor as being both readable and writable, and poll(2) returns a POLLERR event. As noted above, write(2) can never overflow the counter. However an overflow can occur if 2^64 eventfd "signal posts" were per? formed by the KAIO subsystem (theoretically possible, but practically unlikely). If an overflow has occurred, then read(2) will return that maximum uint64_t value (i.e., 0xffffffffffffffff).

The eventfd file descriptor also supports the other file-de? scriptor multiplexing APIs: pselect(2) and ppoll(2).

close(2)

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

A copy of the file descriptor created by eventfd() is inherited by the child produced by fork(2). The duplicate file descriptor is associated with the same eventfd object. File descriptors created by eventfd() are preserved across execve(2), unless the close-on-exec flag has been set.

RETURN VALUE

On success, eventfd() returns a new eventfd file descriptor. On error, -1 is returned and errno is set to indicate the error.

ERRORS

EINVAL An unsupported value was specified in flags.

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 eventfd file de?scriptor.

VERSIONS

eventfd() is available on Linux since kernel 2.6.22. Working support is provided in glibc since version 2.8. The eventfd2() system call (see NOTES) is available on Linux since kernel 2.6.27. Since version 2.9, the glibc eventfd() wrapper will employ the eventfd2() system call, if it is supported by the kernel.

ATTRIBUTES

For an explanation of the terms used in this section, see at?tributes(7).

?????????????????????????????????????????

?Interface ? Attribute ? Value ?

?????????????????????????????????????????

?eventfd() ? Thread safety ? MT-Safe ?

?????????????????????????????????????????

CONFORMING TO

eventfd() and eventfd2() are Linux-specific.

## NOTES

Applications can use an eventfd file descriptor instead of a pipe (see pipe(2)) in all cases where a pipe is used simply to signal events. The kernel overhead of an eventfd file descriptor is much lower than that of a pipe, and only one file descriptor is required (versus the two required for a pipe).

When used in the kernel, an eventfd file descriptor can provide a bridge from kernel to user space, allowing, for example, functionali? ties like KAIO (kernel AIO) to signal to a file descriptor that some operation is complete.

A key point about an eventfd file descriptor is that it can be moni? tored just like any other file descriptor using select(2), poll(2), or epoll(7). This means that an application can simultaneously monitor the readiness of "traditional" files and the readiness of other kernel mechanisms that support the eventfd interface. (Without the eventfd() interface, these mechanisms could not be multiplexed via select(2), poll(2), or epoll(7).)

The current value of an eventfd counter can be viewed via the entry for the corresponding file descriptor in the process's /proc/[pid]/fdinfo directory. See proc(5) for further details.

### C library/kernel differences

There are two underlying Linux system calls: eventfd() and the more re? cent eventfd2(). The former system call does not implement a flags ar? gument. The latter system call implements the flags values described above. The glibc wrapper function will use eventfd2() where it is available.

### Additional glibc features

The GNU C library defines an additional type, and two functions that attempt to abstract some of the details of reading and writing on an eventfd file descriptor:

```
typedef uint64_t eventfd_t;
int eventfd_read(int fd, eventfd_t *value);
```

int eventfd_write(int fd, eventfd_t value);

The functions perform the read and write operations on an eventfd file descriptor, returning 0 if the correct number of bytes was transferred, or -1 otherwise.

EXAMPLES

The following program creates an eventfd file descriptor and then forks to create a child process. While the parent briefly sleeps, the child writes each of the integers supplied in the program's command-line ar? guments to the eventfd file descriptor. When the parent has finished sleeping, it reads from the eventfd file descriptor.

The following shell session shows a sample run of the program:

```
$ ./a.out 1 2 4 7 14
Child writing 1 to efd
Child writing 2 to efd
Child writing 4 to efd
Child writing 7 to efd
Child writing 14 to efd
Child completed write loop
Parent about to read
Parent read 28 (0x1c) from efd
```

Program source

```
#include <sys/eventfd.h>
#include <unistd.h>
#include <inttypes.h>           /* Definition of PRIu64 & PRIx64 */
#include <stdlib.h>
#include <stdio.h>
#include <stdint.h>             /* Definition of uint64_t */
#define handle_error(msg) \
    do { perror(msg); exit(EXIT_FAILURE); } while (0)
int
main(int argc, char *argv[])
{
    int efd;
```

```
    uint64_t u;

    ssize_t s;

    if (argc < 2) {

        fprintf(stderr, "Usage: %s <num>...\n", argv[0]);

        exit(EXIT_FAILURE);

    }

    efd = eventfd(0, 0);

    if (efd == -1)

        handle_error("eventfd");

    switch (fork()) {

    case 0:

        for (int j = 1; j < argc; j++) {

            printf("Child writing %s to efd\n", argv[j]);

            u = strtoull(argv[j], NULL, 0);

                    /* strtoull() allows various bases */

            s = write(efd, &u, sizeof(uint64_t));

            if (s != sizeof(uint64_t))

                handle_error("write");

        }

        printf("Child completed write loop\n");

        exit(EXIT_SUCCESS);

    default:

        sleep(2);

        printf("Parent about to read\n");

        s = read(efd, &u, sizeof(uint64_t));

        if (s != sizeof(uint64_t))

            handle_error("read");

        printf("Parent read %"PRIu64" (%#"PRIx64") from efd\n", u, u);

        exit(EXIT_SUCCESS);

    case -1:

        handle_error("fork");

    }

}
```

## SEE ALSO

futex(2), pipe(2), poll(2), read(2), select(2), signalfd(2), timerfd_create(2), write(2), epoll(7), sem_overview(7)

## COLOPHON

This page is part of release 5.10 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/.