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

$ man timerfd_create.2

NAME

timerfd_create, timerfd_settime, timerfd_gettime - timers that notify via file descriptors

SYNOPSIS

```
#include <sys/timerfd.h>

int timerfd_create(int clockid, int flags);

int timerfd_settime(int fd, int flags,
                    const struct itimerspec *new_value,
                    struct itimerspec *old_value);

int timerfd_gettime(int fd, struct itimerspec *curr_value);
```

DESCRIPTION

These system calls create and operate on a timer that delivers timer expiration notifica? tions via a file descriptor. They provide an alternative to the use of setitimer(2) or timer_create(2), with the advantage that the file descriptor may be monitored by se? lect(2), poll(2), and epoll(7).

The use of these three system calls is analogous to the use of timer_create(2), timer_set? time(2), and timer_gettime(2). (There is no analog of timer_getoverrun(2), since that functionality is provided by read(2), as described below.)

timerfd_create()

timerfd_create() creates a new timer object, and returns a file descriptor that refers to that timer. The clockid argument specifies the clock that is used to mark the progress of the timer, and must be one of the following:

CLOCK_REALTIME

A settable system-wide real-time clock.

CLOCK_MONOTONIC

A nonsettable monotonically increasing clock that measures time from some unspeci?
fied point in the past that does not change after system startup.

CLOCK_BOOTTIME (Since Linux 3.15)

Like CLOCK_MONOTONIC, this is a monotonically increasing clock. However, whereas the CLOCK_MONOTONIC clock does not measure the time while a system is suspended, the CLOCK_BOOTTIME clock does include the time during which the system is sus?
pended. This is useful for applications that need to be suspend-aware. CLOCK_RE?
ALTIME is not suitable for such applications, since that clock is affected by dis?
continuous changes to the system clock.

CLOCK_REALTIME_ALARM (since Linux 3.11)

This clock is like CLOCK_REALTIME, but will wake the system if it is suspended. The caller must have the CAP_WAKE_ALARM capability in order to set a timer against this clock.

CLOCK_BOOTTIME_ALARM (since Linux 3.11)

This clock is like CLOCK_BOOTTIME, but will wake the system if it is suspended. The caller must have the CAP_WAKE_ALARM capability in order to set a timer against this clock.

See clock_getres(2) for some further details on the above clocks.

The current value of each of these clocks can be retrieved using clock_gettime(2).

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

TFD_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 ex?
tra calls to fcntl(2) to achieve the same result.

TFD_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 versions up to and including 2.6.26, flags must be specified as zero.

timerfd_settime()

timerfd_settime() arms (starts) or disarms (stops) the timer referred to by the file de?
scriptor fd.

The new_value argument specifies the initial expiration and interval for the timer. The itimerspec structure used for this argument contains two fields, each of which is in turn a structure of type timespec:

```
struct timespec {
    time_t tv_sec;            /* Seconds */
    long   tv_nsec;           /* Nanoseconds */
};
struct itimerspec {
    struct timespec it_interval;  /* Interval for periodic timer */
    struct timespec it_value;     /* Initial expiration */
};
```

new_value.it_value specifies the initial expiration of the timer, in seconds and nanosec? onds. Setting either field of new_value.it_value to a nonzero value arms the timer. Set? ting both fields of new_value.it_value to zero disarms the timer.

Setting one or both fields of new_value.it_interval to nonzero values specifies the pe? riod, in seconds and nanoseconds, for repeated timer expirations after the initial expira? tion. If both fields of new_value.it_interval are zero, the timer expires just once, at the time specified by new_value.it_value.

By default, the initial expiration time specified in new_value is interpreted relative to the current time on the timer's clock at the time of the call (i.e., new_value.it_value specifies a time relative to the current value of the clock specified by clockid). An ab? solute timeout can be selected via the flags argument.

The flags argument is a bit mask that can include the following values:

TFD_TIMER_ABSTIME

Interpret new_value.it_value as an absolute value on the timer's clock. The timer will expire when the value of the timer's clock reaches the value specified in new_value.it_value.

TFD_TIMER_CANCEL_ON_SET

If this flag is specified along with TFD_TIMER_ABSTIME and the clock for this timer is CLOCK_REALTIME or CLOCK_REALTIME_ALARM, then mark this timer as cancelable if the real-time clock undergoes a discontinuous change (settimeofday(2), clock_set? time(2), or similar). When such changes occur, a current or future read(2) from the file descriptor will fail with the error ECANCELED.

If the old_value argument is not NULL, then the itimerspec structure that it points to is used to return the setting of the timer that was current at the time of the call; see the description of timerfd_gettime() following.

timerfd_gettime()

timerfd_gettime() returns, in curr_value, an itimerspec structure that contains the cur?rent setting of the timer referred to by the file descriptor fd.

The it_value field returns the amount of time until the timer will next expire. If both fields of this structure are zero, then the timer is currently disarmed. This field al?ways contains a relative value, regardless of whether the TFD_TIMER_ABSTIME flag was spec?ified when setting the timer.

The it_interval field returns the interval of the timer. If both fields of this structure are zero, then the timer is set to expire just once, at the time specified by curr_value.it_value.

Operating on a timer file descriptor

The file descriptor returned by timerfd_create() supports the following additional opera?tions:

read(2)

If the timer has already expired one or more times since its settings were last modified using timerfd_settime(), or since the last successful read(2), then the buffer given to read(2) returns an unsigned 8-byte integer (uint64_t) containing the number of expirations that have occurred. (The returned value is in host byte order?that is, the native byte order for integers on the host machine.)

If no timer expirations have occurred at the time of the read(2), then the call ei?ther blocks until the next timer expiration, or fails with the error EAGAIN if the file descriptor has been made nonblocking (via the use of the fcntl(2) F_SETFL op?eration to set the O_NONBLOCK flag).

A read(2) fails with the error EINVAL if the size of the supplied buffer is less than 8 bytes.

If the associated clock is either CLOCK_REALTIME or CLOCK_REALTIME_ALARM, the timer is absolute (TFD_TIMER_ABSTIME), and the flag TFD_TIMER_CANCEL_ON_SET was specified when calling timerfd_settime(), then read(2) fails with the error ECANCELED if the real-time clock undergoes a discontinuous change. (This allows the reading appli?cation to discover such discontinuous changes to the clock.)

If the associated clock is either CLOCK_REALTIME or CLOCK_REALTIME_ALARM, the timer is absolute (TFD_TIMER_ABSTIME), and the flag TFD_TIMER_CANCEL_ON_SET was not spec? ified when calling timerfd_settime(), then a discontinuous negative change to the clock (e.g., clock_settime(2)) may cause read(2) to unblock, but return a value of 0 (i.e., no bytes read), if the clock change occurs after the time expired, but be? fore the read(2) on the file descriptor.

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 timer expirations have occurred.

The file descriptor also supports the other file-descriptor multiplexing APIs: pse? lect(2), ppoll(2), and epoll(7).

ioctl(2)

The following timerfd-specific command is supported:

TFD_IOC_SET_TICKS (since Linux 3.17)

Adjust the number of timer expirations that have occurred. The argument is a pointer to a nonzero 8-byte integer (uint64_t*) containing the new number of expirations. Once the number is set, any waiter on the timer is woken up. The only purpose of this command is to restore the expirations for the purpose of checkpoint/restore. This operation is available only if the ker? nel was configured with the CONFIG_CHECKPOINT_RESTORE option.

close(2)

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

fork(2) semantics

After a fork(2), the child inherits a copy of the file descriptor created by timerfd_cre? ate(). The file descriptor refers to the same underlying timer object as the correspond? ing file descriptor in the parent, and read(2)s in the child will return information about expirations of the timer.

execve(2) semantics

A file descriptor created by timerfd_create() is preserved across execve(2), and continues to generate timer expirations if the timer was armed.

RETURN VALUE

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

timerfd_settime() and timerfd_gettime() return 0 on success; on error they return -1, and set errno to indicate the error.

ERRORS

timerfd_create() can fail with the following errors:

EINVAL The clockid is not valid.

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 kernel memory to create the timer.

EPERM clockid was CLOCK_REALTIME_ALARM or CLOCK_BOOTTIME_ALARM but the caller did not have the CAP_WAKE_ALARM capability.

timerfd_settime() and timerfd_gettime() can fail with the following errors:

EBADF fd is not a valid file descriptor.

EFAULT new_value, old_value, or curr_value is not valid a pointer.

EINVAL fd is not a valid timerfd file descriptor.

timerfd_settime() can also fail with the following errors:

ECANCELED

See NOTES.

EINVAL new_value is not properly initialized (one of the tv_nsec falls outside the range zero to 999,999,999).

EINVAL flags is invalid.

VERSIONS

These system calls are available on Linux since kernel 2.6.25. Library support is pro? vided by glibc since version 2.8.

CONFORMING TO

These system calls are Linux-specific.

NOTES

Suppose the following scenario for CLOCK_REALTIME or CLOCK_REALTIME_ALARM timer that was created with timerfd_create():

(a) The timer has been started (timerfd_settime()) with the TFD_TIMER_ABSTIME and

TFD_TIMER_CANCEL_ON_SET flags;

(b) A discontinuous change (e.g., settimeofday(2)) is subsequently made to the CLOCK_REAL?

TIME clock; and

(c) the caller once more calls timerfd_settime() to rearm the timer (without first doing a

read(2) on the file descriptor).

In this case the following occurs:

? The timerfd_settime() returns -1 with errno set to ECANCELED. (This enables the caller

to know that the previous timer was affected by a discontinuous change to the clock.)

? The timer is successfully rearmed with the settings provided in the second timerfd_set?

time() call. (This was probably an implementation accident, but won't be fixed now, in

case there are applications that depend on this behaviour.)

BUGS

Currently, timerfd_create() supports fewer types of clock IDs than timer_create(2).

EXAMPLES

The following program creates a timer and then monitors its progress. The program accepts up to three command-line arguments. The first argument specifies the number of seconds for the initial expiration of the timer. The second argument specifies the interval for the timer, in seconds. The third argument specifies the number of times the program should allow the timer to expire before terminating. The second and third command-line arguments are optional.

The following shell session demonstrates the use of the program:

```
$ a.out 3 1 100
0.000: timer started
3.000: read: 1; total=1
4.000: read: 1; total=2
^Z                  # type control-Z to suspend the program
[1]+  Stopped                 ./timerfd3_demo 3 1 100
$ fg                # Resume execution after a few seconds
a.out 3 1 100
9.660: read: 5; total=7
10.000: read: 1; total=8
11.000: read: 1; total=9
^C                  # type control-C to suspend the program
```

Program source

```
#include <sys/timerfd.h>
#include <time.h>
#include <unistd.h>
#include <inttypes.h>      /* Definition of PRIu64 */
#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)

static void
print_elapsed_time(void)
{
    static struct timespec start;
    struct timespec curr;
    static int first_call = 1;
    int secs, nsecs;

    if (first_call) {
        first_call = 0;
        if (clock_gettime(CLOCK_MONOTONIC, &start) == -1)
            handle_error("clock_gettime");
    }

    if (clock_gettime(CLOCK_MONOTONIC, &curr) == -1)
        handle_error("clock_gettime");

    secs = curr.tv_sec - start.tv_sec;
    nsecs = curr.tv_nsec - start.tv_nsec;
    if (nsecs < 0) {
        secs--;
        nsecs += 1000000000;
    }
    printf("%d.%03d: ", secs, (nsecs + 500000) / 1000000);
}

int
```

```
main(int argc, char *argv[])
{
    struct itimerspec new_value;
    int max_exp, fd;
    struct timespec now;
    uint64_t exp, tot_exp;
    ssize_t s;
    if ((argc != 2) && (argc != 4)) {
        fprintf(stderr, "%s init-secs [interval-secs max-exp]\n",
                argv[0]);
        exit(EXIT_FAILURE);
    }
    if (clock_gettime(CLOCK_REALTIME, &now) == -1)
        handle_error("clock_gettime");
    /* Create a CLOCK_REALTIME absolute timer with initial
       expiration and interval as specified in command line */
    new_value.it_value.tv_sec = now.tv_sec + atoi(argv[1]);
    new_value.it_value.tv_nsec = now.tv_nsec;
    if (argc == 2) {
        new_value.it_interval.tv_sec = 0;
        max_exp = 1;
    } else {
        new_value.it_interval.tv_sec = atoi(argv[2]);
        max_exp = atoi(argv[3]);
    }
    new_value.it_interval.tv_nsec = 0;
    fd = timerfd_create(CLOCK_REALTIME, 0);
    if (fd == -1)
        handle_error("timerfd_create");
    if (timerfd_settime(fd, TFD_TIMER_ABSTIME, &new_value, NULL) == -1)
        handle_error("timerfd_settime");
    print_elapsed_time();
    printf("timer started\n");
```

```
        for (tot_exp = 0; tot_exp < max_exp;) {
            s = read(fd, &exp, sizeof(uint64_t));
            if (s != sizeof(uint64_t))
                handle_error("read");
            tot_exp += exp;
            print_elapsed_time();
            printf("read: %" PRIu64 "; total=%" PRIu64 "\n", exp, tot_exp);
        }
        exit(EXIT_SUCCESS);
    }
```

SEE ALSO

eventfd(2), poll(2), read(2), select(2), setitimer(2), signalfd(2), timer_create(2), timer_gettime(2), timer_settime(2), epoll(7), time(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/.