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 notifications 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 **select**(2), **poll**(2), and **epoll**(7).

The use of these three system calls is analogous to the use of **timer_create**(2), **timer_settime**(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 either **CLOCK_REALTIME** or **CLOCK_MONOTONIC**. **CLOCK_REALTIME** is a settable system-wide clock. **CLOCK_MONOTONIC** is a non-settable clock that is not affected by discontinuous changes in the system clock (e.g., manual changes to system time). 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 new open file description. Using this flag saves extra 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 descriptor *fd*.

The *new_value* argument specifies the initial expiration and interval for the timer. The *itimer* 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 nanoseconds. Setting either field of *new_value.it_value* to a non-zero value arms the timer. Setting both fields of *new_value.it_value* to zero disarms the timer.

Setting one or both fields of *new_value.it_interval* to non-zero values specifies the period, in seconds and nanoseconds, for repeated timer expirations after the initial expiration. If both fields of *new_value.it_interval* are zero, the timer expires just once, at the time specified by *new_value.it_value*.

The *flags* argument is either 0, to start a relative timer (*new_value.it_interval* specifies a time relative to the current value of the clock specified by *clockid*), or **TFD_TIMER_ABSTIME**, to start an absolute timer (*new_value.it_value* specifies an absolute time for the clock specified by *clockid*; that is, the timer will expire when the value of that clock reaches the value specified in *new_value.it_value*).

The *old_value* argument returns a structure containing 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 current 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 always contains a relative value, regardless of whether the **TFD_TIMER_ABSTIME** flag was specified 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 operations:

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, i.e., 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 either blocks until the next timer expiration, or fails with the error **EAGAIN** if the file descriptor has been made non-blocking (via the use of the **fcntl**(2) **F_SETFL** operation to set the **O_NONBLOCK** flag).

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

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: **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 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_create**(). The file descriptor refers to the same underlying timer object as the corresponding 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* argument is neither **CLOCK_MONOTONIC** nor **CLOCK_REALTIME**;

EINVAL
flags is invalid; or, in Linux 2.6.26 or earlier, *flags* is non-zero.

EMFILE
The per-process limit 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.

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:

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 provided by glibc since version 2.8.

CONFORMING TO

These system calls are Linux-specific.

EXAMPLE

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 <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: %llu; total=%llu\n",
                (unsigned long long) exp,
                (unsigned long long) tot_exp);
    }

    exit(EXIT_SUCCESS);
}
```

BUGS

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

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 3.22 of the Linux *man-pages* project. A description of the project, and information about reporting bugs, can be found at http://www.kernel.org/doc/man-pages/.