NAME

timer_create − create a POSIX per-process timer

LIBRARY

Real-time library (*librt*, *−lrt*)

SYNOPSIS

```
#include <signal.h>          /* Definition of SIGEV_* constants */
#include <time.h>

int timer_create(clockid_t clockid,
                 struct sigevent *_Nullable restrict sevp,
                 timer_t *restrict timerid);
```

Feature Test Macro Requirements for glibc (see **feature_test_macros**(7)):

timer_create():
  _POSIX_C_SOURCE >= 199309L

DESCRIPTION

timer_create() creates a new per-process interval timer. The ID of the new timer is returned in the buffer pointed to by *timerid*, which must be a non-null pointer. This ID is unique within the process, until the timer is deleted. The new timer is initially disarmed.

The *clockid* argument specifies the clock that the new timer uses to measure time. It can be specified as one of the following values:

CLOCK_REALTIME
A settable system-wide real-time clock.

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

CLOCK_PROCESS_CPUTIME_ID (since Linux 2.6.12)
A clock that measures (user and system) CPU time consumed by (all of the threads in) the calling process.

CLOCK_THREAD_CPUTIME_ID (since Linux 2.6.12)
A clock that measures (user and system) CPU time consumed by the calling thread.

CLOCK_BOOTTIME (Since Linux 2.6.39)
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 suspended. This is useful for applications that need to be suspend-aware. **CLOCK_REALTIME** is not suitable for such applications, since that clock is affected by discontinuous changes to the system clock.

CLOCK_REALTIME_ALARM (since Linux 3.0)
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 3.0)
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.

CLOCK_TAI (since Linux 3.10)
A system-wide clock derived from wall-clock time but ignoring leap seconds.

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

As well as the above values, *clockid* can be specified as the *clockid* returned by a call to **clock_getcpuclockid**(3) or **pthread_getcpuclockid**(3).

The *sevp* argument points to a *sigevent* structure that specifies how the caller should be notified when the timer expires. For the definition and general details of this structure, see **sigevent**(7).

The *sevp.sigev_notify* field can have the following values:

SIGEV_NONE
: Don't asynchronously notify when the timer expires. Progress of the timer can be monitored using **timer_gettime**(2).

SIGEV_SIGNAL
: Upon timer expiration, generate the signal *sigev_signo* for the process. See **sigevent**(7) for general details. The *si_code* field of the *siginfo_t* structure will be set to **SI_TIMER**. At any point in time, at most one signal is queued to the process for a given timer; see **timer_getoverrun**(2) for more details.

SIGEV_THREAD
: Upon timer expiration, invoke *sigev_notify_function* as if it were the start function of a new thread. See **sigevent**(7) for details.

SIGEV_THREAD_ID (Linux-specific)
: As for **SIGEV_SIGNAL**, but the signal is targeted at the thread whose ID is given in *sigev_notify_thread_id*, which must be a thread in the same process as the caller. The*sig ev_notify_thread_id* field specifies a kernel thread ID, that is, the value returned by **clone**(2) or **gettid**(2). This flag is intended only for use by threading libraries.

Specifying *sevp* as NULL is equivalent to specifying a pointer to a *sigevent* structure in which *sigev_notify* is **SIGEV_SIGNAL**, *sigev_signo* is **SIGALRM**, and *sigev_value.sival_int* is the timer ID.

RETURN VALUE

On success, **timer_create**() returns 0, and the ID of the new timer is placed in**timerid*. On f ailure, −1 is returned, and *errno* is set to indicate the error.

ERRORS

EAGAIN
: Temporary error during kernel allocation of timer structures.

EINVAL
: Clock ID, *sigev_notify*, *sigev_signo*, or *sigev_notify_thread_id* is invalid.

ENOMEM
: Could not allocate memory.

ENOTSUP
: The kernel does not support creating a timer against this *clockid*.

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

VERSIONS

This system call is available since Linux 2.6.

STANDARDS

POSIX.1-2001, POSIX.1-2008.

NOTES

A program may create multiple interval timers using **timer_create**().

Timers are not inherited by the child of a **fork**(2), and are disarmed and deleted during an **execve**(2).

The kernel preallocates a "queued real-time signal" for each timer created using **timer_create**(). Consequently, the number of timers is limited by the **RLIMIT_SIGPENDING** resource limit (see **setrlimit**(2)).

The timers created by **timer_create**() are commonly known as "POSIX (interval) timers". The POSIX timers API consists of the following interfaces:

timer_create()
: Create a timer.

timer_settime(2)
: Arm (start) or disarm (stop) a timer.

timer_gettime(2)
: Fetch the time remaining until the next expiration of a timer, along with the interval setting of the timer.

timer_getoverrun(2)
: Return the overrun count for the last timer expiration.

timer_delete(2)
: Disarm and delete a timer.

Since Linux 3.10, the */proc/*pid*/timers* file can be used to list the POSIX timers for the process with PID *pid*. See**pr oc**(5) for further information.

Since Linux 4.10, support for POSIX timers is a configurable option that is enabled by default. Kernel support can be disabled via the **CONFIG_POSIX_TIMERS** option.

C library/kernel differences

Part of the implementation of the POSIX timers API is provided by glibc. In particular:

- Much of the functionality for **SIGEV_THREAD** is implemented within glibc, rather than the kernel. (This is necessarily so, since the thread involved in handling the notification is one that must be managed by the C library POSIX threads implementation.) Although the notification delivered to the process is via a thread, internally the NPTL implementation uses a *sigev_notify* value of **SIGEV_THREAD_ID** along with a real-time signal that is reserved by the implementation (see **nptl**(7)).
- The implementation of the default case where *evp* is NULL is handled inside glibc, which invokes the underlying system call with a suitably populated *sigevent* structure.
- The timer IDs presented at user level are maintained by glibc, which maps these IDs to the timer IDs employed by the kernel.

The POSIX timers system calls first appeared in Linux 2.6. Prior to this, glibc provided an incomplete user-space implementation (**CLOCK_REALTIME** timers only) using POSIX threads, and before glibc 2.17, the implementation falls back to this technique on systems running kernels older than Linux 2.6.

EXAMPLES

The program below takes two arguments: a sleep period in seconds, and a timer frequency in nanoseconds. The program establishes a handler for the signal it uses for the timer, blocks that signal, creates and arms a timer that expires with the given frequency, sleeps for the specified number of seconds, and then unblocks the timer signal. Assuming that the timer expired at least once while the program slept, the signal handler will be invoked, and the handler displays some information about the timer notification. The program terminates after one invocation of the signal handler.

In the following example run, the program sleeps for 1 second, after creating a timer that has a frequency of 100 nanoseconds. By the time the signal is unblocked and delivered, there have been around ten million overruns.

```
$ ./a.out 1 100
Establishing handler for signal 34
Blocking signal 34
timer ID is 0x804c008
Sleeping for 1 seconds
Unblocking signal 34
Caught signal 34
    sival_ptr = 0xbfb174f4;     *sival_ptr = 0x804c008
    overrun count = 10004886
```

Program source

```
#include <signal.h>
#include <stdint.h>
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <unistd.h>

#define CLOCKID CLOCK_REALTIME
#define SIG SIGRTMIN

#define errExit(msg)    do { perror(msg); exit(EXIT_FAILURE); \
                        } while (0)

static void
print_siginfo(siginfo_t *si)
{
    int      or;
    timer_t  *tidp;

    tidp = si−>si_value.sival_ptr;

    printf("    sival_ptr = %p; ", si−>si_value.sival_ptr);
    printf("    *sival_ptr = %#jx\n", (uintmax_t) *tidp);

    or = timer_getoverrun(*tidp);
    if (or == −1)
        errExit("timer_getoverrun");
    else
        printf("    overrun count = %d\n", or);
}

static void
handler(int sig, siginfo_t *si, void *uc)
{
    /* Note: calling printf() from a signal handler is not safe
       (and should not be done in production programs), since
       printf() is not async−signal−safe; see signal−safety(7).
       Nevertheless, we use printf() here as a simple way of
       showing that the handler was called. */

    printf("Caught signal %d\n", sig);
    print_siginfo(si);
    signal(sig, SIG_IGN);
}

int
main(int argc, char *argv[])
{
    timer_t            timerid;
    sigset_t           mask;
    long long          freq_nanosecs;
    struct sigevent    sev;
    struct sigaction   sa;
    struct itimerspec  its;

    if (argc != 3) {
        fprintf(stderr, "Usage: %s <sleep-secs> <freq-nanosecs>\n",
                argv[0]);
        exit(EXIT_FAILURE);
    }

    /* Establish handler for timer signal. */

    printf("Establishing handler for signal %d\n", SIG);
    sa.sa_flags = SA_SIGINFO;
    sa.sa_sigaction = handler;
    sigemptyset(&sa.sa_mask);
    if (sigaction(SIG, &sa, NULL) == -1)
        errExit("sigaction");

    /* Block timer signal temporarily. */

    printf("Blocking signal %d\n", SIG);
    sigemptyset(&mask);
    sigaddset(&mask, SIG);
    if (sigprocmask(SIG_SETMASK, &mask, NULL) == -1)
        errExit("sigprocmask");

    /* Create the timer. */

    sev.sigev_notify = SIGEV_SIGNAL;
    sev.sigev_signo = SIG;
    sev.sigev_value.sival_ptr = &timerid;
    if (timer_create(CLOCKID, &sev, &timerid) == -1)
        errExit("timer_create");

    printf("timer ID is %#jx\n", (uintmax_t) timerid);

    /* Start the timer. */

    freq_nanosecs = atoll(argv[2]);
    its.it_value.tv_sec = freq_nanosecs / 1000000000;
    its.it_value.tv_nsec = freq_nanosecs % 1000000000;
    its.it_interval.tv_sec = its.it_value.tv_sec;
    its.it_interval.tv_nsec = its.it_value.tv_nsec;

    if (timer_settime(timerid, 0, &its, NULL) == -1)
         errExit("timer_settime");

    /* Sleep for a while; meanwhile, the timer may expire
       multiple times. */

    printf("Sleeping for %d seconds\n", atoi(argv[1]));
    sleep(atoi(argv[1]));

    /* Unlock the timer signal, so that timer notification
```

```
               can be delivered. */

            printf("Unblocking signal %d\n", SIG);
            if (sigprocmask(SIG_UNBLOCK, &mask, NULL) == −1)
                errExit("sigprocmask");

            exit(EXIT_SUCCESS);
        }
```

SEE ALSO

clock_gettime(2), **setitimer**(2), **timer_delete**(2), **timer_getoverrun**(2), **timer_settime**(2), **timerfd_create**(2), **clock_getcpuclockid**(3), **pthread_getcpuclockid**(3), **pthreads**(7), **sigevent**(7), **signal**(7), **time**(7)