NAME

timer_create − create a POSIX per-process timer

SYNOPSIS

#include <signal.h>
#include <time.h>

int timer_create(clockid_t *clockid***, struct sigevent ***evp***,**
timer_t ***timerid*);**

Link with *−lrt*.

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

timer_create(): _POSIX_C_SOURCE >= 199309

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 non-settable 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.

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 *evp* argument points to a *sigevent* structure that specifies how the caller should be notified when the timer expires. This structure is defined something like the following:

```
union sigval {
    int   sival_int;
    void *sival_ptr;
};

struct sigevent {
    int          sigev_notify;    /* Notification method */
    int          sigev_signo;     /* Timer expiration signal */
    union sigval sigev_value;     /* Value accompanying signal or
                                     passed to thread function */
    void       (*sigev_notify_function) (union sigval);
                 /* Function used for thread
                    notifications (SIGEV_THREAD) */
    void        *sigev_notify_attributes;
                 /* Attributes for notification thread
```

```
                       (SIGEV_THREAD) */
        pid_t      sigev_notify_thread_id;
                      /* ID of thread to signal (SIGEV_THREAD_ID) */
    };
```

Some of these fields may be defined as part of a union: a program should only employ those fields relevant to the value specified in *sigev_notify*. This 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. If *sigev_signo* is a real-time signal, then it will be accompanied by the data specified in *sigev_value* (like the signal-accompanying data for **sigqueue**(2)). 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. (Among the implementation possibilities here are that each timer notification could result in the creation of a new thread, or that a single thread is created to receive all notifications.) The function is invoked with *sigev_value* as its sole argument. If *sigev_notify_attributes* is not NULL, it should point to a *pthread_attr_t* structure that defines attributes for the new thread (see **pthread_attr_init**(3).

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 *sigev_notify_thread_id* field specifies a kernel thread ID, that is, the value returned by **clone**(2) or **gettid**(2). This flag is only intended for use by threading libraries.

Specifying *evp* 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 failure, −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*, *sigev_notify_thread_id* is invalid.

ENOMEM
Could not allocate memory.

VERSIONS

This system call is available since Linux 2.6.

CONFORMING TO

POSIX.1-2001

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.

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

* The functionality for **SIGEV_THREAD** is implemented within glibc, rather than the kernel.
* 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 userspace implementation (**CLOCK_REALTIME** timers only) using POSIX threads, and current glibc falls back to this implementation on systems running pre-2.6 Linux kernels.

EXAMPLE

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 10
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 <stdlib.h>
#include <unistd.h>
#include <stdio.h>
#include <signal.h>
#include <time.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)
{
    timer_t *tidp;
    int or;

    tidp = si−>si_value.sival_ptr;

    printf("    sival_ptr = %p; ", si−>si_value.sival_ptr);
    printf("    *sival_ptr = 0x%lx\n", (long) *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
       strictly correct, since printf() is not async−signal−safe;
       see signal(7) */

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

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

    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 0x%lx\n", (long) 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_settime**(2), **timer_getoverrun**(2), **timerfd_create**(2), **clock_getcpuclockid**(3), **pthread_getcpuclockid**(3), **pthreads**(7), **signal**(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/.