23

TIMERS AND SLEEPING

A timer allows a process to schedule a notification for itself to occur at some time in the future. Sleeping allows a process (or thread) to suspend execution for a period of time. This chapter describes the interfaces used for setting timers and for sleeping. It covers the following topics:

- the classical UNIX APIs for setting interval timers (*setitimer()* and *alarm()*) to notify a process when a certain amount of time has passed;
- the APIs that allow a process to sleep for a specified interval;
- the POSIX.1b clocks and timers APIs; and
- the Linux-specific *timerfd* facility, which allows the creation of timers whose expirations can be read from a file descriptor.

23.1 Interval Timers

The *setitimer()* system call establishes an *interval timer*, which is a timer that expires at a future point in time and (optionally) at regular intervals after that.

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

int setitimer(int which, const struct itimerval *new_value,
              struct itimerval *old_value);
                                                Returns 0 on success, or –1 on error
```

Using *setitimer()*, a process can establish three different types of timers, by specifying *which* as one of the following:

ITIMER_REAL
: Create a timer that counts down in real (i.e., wall clock) time. When the timer expires, a SIGALRM signal is generated for the process.

ITIMER_VIRTUAL
: Create a timer that counts down in process virtual time (i.e., user-mode CPU time). When the timer expires, a SIGVTALRM signal is generated for the process.

ITIMER_PROF
: Create a *profiling* timer. A profiling timer counts in process time (i.e., the sum of both user-mode and kernel-mode CPU time). When the timer expires, a SIGPROF signal is generated for the process.

The default disposition of all of the timer signals is to terminate the process. Unless this is the desired result, we must to establish a handler for the signal delivered by the timer.

The *new_value* and *old_value* arguments are pointers to *itimerval* structures, defined as follows:

```
struct itimerval {
    struct timeval it_interval;     /* Interval for periodic timer */
    struct timeval it_value;        /* Current value (time until
                                       next expiration) */
};
```

Each of the fields in the *itimerval* structure is in turn a structure of type *timeval*, containing seconds and microseconds fields:

```
struct timeval {
    time_t      tv_sec;             /* Seconds */
    suseconds_t tv_usec;            /* Microseconds (long int) */
};
```

The *it_value* substructure of the *new_value* argument specifies the delay until the timer is to expire. The *it_interval* substructure specifies whether this is to be a periodic timer. If both fields of *it_interval* are set to 0, then the timer is expires just once, at the time given by *it_value*. If one or both of the *it_interval* fields are nonzero, then, after each expiration of the timer, the timer will be reset to expire again at the specified interval.

A process has only one of each of the three types of timers. If we call *setitimer()* a second time, it will change the characteristics of any existing timer corresponding

to *which*. If we call *setitimer()* with both fields of *new_value.it_value* set to 0, then any existing timer is disabled.

If *old_value* is not NULL, then it points to an *itimerval* structure that is used to return the previous value of the timer. If both fields of *old_value.it_value* are 0, then the timer was previously disabled. If both fields of *old_value.it_interval* are 0, then the previous timer was set to expire just once, at the time given by *old_value.it_value*. Retrieving the previous settings of the timer can be useful if we want to restore the settings after the new timer has expired. If we are not interested in the previous value of the timer, we can specify *old_value* as NULL.

As a timer progresses, it counts down from the initial value (*it_value*) toward 0. When the timer reaches 0, the corresponding signal is sent to the process, and then, if the interval (*it_interval*) is nonzero, the timer value (*it_value*) is reloaded, and counting down toward 0 recommences.

At any time, we can use *getitimer()* to retrieve the current state of the timer in order to see how much time is left before it next expires.

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

int getitimer(int which, struct itimerval *curr_value);
```

Returns 0 on success, or −1 on error

The *getitimer()* system call returns the current state of the timer specified by *which*, in the buffer pointed to by *curr_value*. This is exactly the same information as is returned via the *old_value* argument of *setitimer()*, with the difference that we don't need to change the timer settings in order to retrieve the information. The *curr_value.it_value* substructure returns the amount of time remaining until the timer next expires. This value changes as the timer counts down, and is reset on timer expiration if a nonzero *it_interval* value was specified when the timer was set. The *curr_value.it_interval* substructure returns the interval for this timer; this value remains unchanged until a subsequent call to *setitimer()*.

Timers established using *setitimer()* (and *alarm()*, which we discuss shortly) are preserved across *exec()*, but are not inherited by a child created by *fork()*.

> SUSv4 marks *getitimer()* and *setitimer()* obsolete, noting that the POSIX timers API (Section 23.6) is preferred.

Example program

Listing 23-1 demonstrates the use of *setitimer()* and *getitimer()*. This program performs the following steps:

- Establish a handler for the SIGALRM signal ③.
- Set the value and interval fields for a real (ITIMER_REAL) timer using the values supplied in its command-line arguments ④. If these arguments are absent, the program sets a timer that expires just once, after 2 seconds.
- Execute a continuous loop ⑤, consuming CPU time and periodically calling the function *displayTimes()* ①, which displays the elapsed real time since the program began, as well as the current state of the ITIMER_REAL timer.

Each time the timer expires, the SIGALRM handler is invoked, and it sets a global flag, *gotAlarm* ②. Whenever this flag is set, the loop in the main program calls *displayTimes()* in order to show when the handler was called and the state of the timer ⑥. (We designed the signal handler in this manner to avoid calling non-async-signal-functions from within the handler, for the reasons described in Section 21.1.2.) If the timer has a zero interval, then the program exits on delivery of the first signal; otherwise, it catches up to three signals before terminating ⑦.

When we run the program in Listing 23-1, we see the following:

```
$ ./real_timer 1 800000 1 0          Initial value 1.8 seconds, interval 1 second
       Elapsed   Value  Interval
START:    0.00
Main:     0.50    1.30    1.00       Timer counts down until expiration
Main:     1.00    0.80    1.00
Main:     1.50    0.30    1.00
ALARM:    1.80    1.00    1.00       On expiration, timer is reloaded from interval
Main:     2.00    0.80    1.00
Main:     2.50    0.30    1.00
ALARM:    2.80    1.00    1.00
Main:     3.00    0.80    1.00
Main:     3.50    0.30    1.00
ALARM:    3.80    1.00    1.00
That's all folks
```

Listing 23-1: Using a real-time timer

timers/real_timer.c

```
#include <signal.h>
#include <sys/time.h>
#include <time.h>
#include "tlpi_hdr.h"

static volatile sig_atomic_t gotAlarm = 0;
                        /* Set nonzero on receipt of SIGALRM */

/* Retrieve and display the real time, and (if 'includeTimer' is
   TRUE) the current value and interval for the ITIMER_REAL timer */

static void
① displayTimes(const char *msg, Boolean includeTimer)
{
    struct itimerval itv;
    static struct timeval start;
    struct timeval curr;
    static int callNum = 0;             /* Number of calls to this function */

    if (callNum == 0)                   /* Initialize elapsed time meter */
        if (gettimeofday(&start, NULL) == -1)
            errExit("gettimeofday");

    if (callNum % 20 == 0)              /* Print header every 20 lines */
        printf("       Elapsed   Value Interval\n");
```

```
    if (gettimeofday(&curr, NULL) == -1)
        errExit("gettimeofday");
    printf("%-7s %6.2f", msg, curr.tv_sec - start.tv_sec +
                        (curr.tv_usec - start.tv_usec) / 1000000.0);

    if (includeTimer) {
        if (getitimer(ITIMER_REAL, &itv) == -1)
            errExit("getitimer");
        printf("  %6.2f  %6.2f",
                itv.it_value.tv_sec + itv.it_value.tv_usec / 1000000.0,
                itv.it_interval.tv_sec + itv.it_interval.tv_usec / 1000000.0);
    }

    printf("\n");
    callNum++;
}

static void
sigalrmHandler(int sig)
{
②   gotAlarm = 1;
}

int
main(int argc, char *argv[])
{
    struct itimerval itv;
    clock_t prevClock;
    int maxSigs;                /* Number of signals to catch before exiting */
    int sigCnt;                 /* Number of signals so far caught */
    struct sigaction sa;

    if (argc > 1 && strcmp(argv[1], "--help") == 0)
        usageErr("%s [secs [usecs [int-secs [int-usecs]]]]\n", argv[0]);

    sigCnt = 0;

    sigemptyset(&sa.sa_mask);
    sa.sa_flags = 0;
    sa.sa_handler = sigalrmHandler;
③   if (sigaction(SIGALRM, &sa, NULL) == -1)
        errExit("sigaction");

    /* Exit after 3 signals, or on first signal if interval is 0 */

    maxSigs = (itv.it_interval.tv_sec == 0 &&
                itv.it_interval.tv_usec == 0) ? 1 : 3;

    displayTimes("START:", FALSE);

    /* Set timer from the command-line arguments */

    itv.it_value.tv_sec = (argc > 1) ? getLong(argv[1], 0, "secs") : 2;
    itv.it_value.tv_usec = (argc > 2) ? getLong(argv[2], 0, "usecs") : 0;
```

```
    itv.it_interval.tv_sec = (argc > 3) ? getLong(argv[3], 0, "int-secs") : 0;
    itv.it_interval.tv_usec = (argc > 4) ? getLong(argv[4], 0, "int-usecs") : 0;

④   if (setitimer(ITIMER_REAL, &itv, 0) == -1)
        errExit("setitimer");

    prevClock = clock();
    sigCnt = 0;

⑤   for (;;) {

        /* Inner loop consumes at least 0.5 seconds CPU time */

        while (((clock() - prevClock) * 10 / CLOCKS_PER_SEC) < 5) {
⑥           if (gotAlarm) {                     /* Did we get a signal? */
                gotAlarm = 0;
                displayTimes("ALARM:", TRUE);

                sigCnt++;
⑦               if (sigCnt >= maxSigs) {
                    printf("That's all folks\n");
                    exit(EXIT_SUCCESS);
                }
            }
        }

        prevClock = clock();
        displayTimes("Main: ", TRUE);
    }
}
```

timers/real_timer.c

A simpler timer interface: *alarm()*

The *alarm()* system call provides a simple interface for establishing a real-time timer that expires once, with no repeating interval. (Historically, *alarm()* was the original UNIX API for setting a timer.)

```
#include <unistd.h>

unsigned int alarm(unsigned int seconds);
```

Always succeeds, returning number of seconds remaining on any previously set timer, or 0 if no timer previously was set

The *seconds* argument specifies the number of seconds in the future when the timer is to expire. At that time, a SIGALRM signal is delivered to the calling process.

Setting a timer with *alarm()* overrides any previously set timer. We can disable an existing timer using the call *alarm(0)*.

As its return value, *alarm()* gives us the number of seconds remaining until the expiration of any previously set timer, or 0 if no timer was set.

An example of the use of *alarm()* is shown in Section 23.3.

> In some later example programs in this book, we use *alarm()* to start a timer without establishing a corresponding `SIGALRM` handler, as a technique for ensuring that a process is killed if it is not otherwise terminated.

Interactions between *setitimer()* and *alarm()*

On Linux, *alarm()* and *setitimer()* share the same per-process real-time timer, which means that setting a timer with one of these functions changes any timer previously set by either of the functions. This may not be the case on other UNIX implementations (i.e., these functions could control independent timers). SUSv3 explicitly leaves unspecified the interactions between *setitimer()* and *alarm()*, as well as the interactions of these functions with the *sleep()* function described in Section 23.4.1. For maximum portability, we should ensure that our applications use only one of *setitimer()* and *alarm()* for setting real-time timers.

23.2 Scheduling and Accuracy of Timers

Depending on system load and the scheduling of processes, a process may not be scheduled to run until some short time (i.e., usually some small fraction of a second) after actual expiration of the timer. Notwithstanding this, the expiration of a periodic timer established by *setitimer()*, or the other interfaces described later in this chapter, will remain regular. For example, if a real-time timer is set to expire every 2 seconds, then the delivery of individual timer events may be subject to the delays just described, but the scheduling of subsequent expirations will nevertheless be at exactly the next 2-second interval. In other words, interval timers are not subject to creeping errors.

Although the *timeval* structure used by *setitimer()* allows for microsecond precision, the accuracy of a timer has traditionally been limited by the frequency of the software clock (Section 10.6). If a timer value does not exactly match a multiple of the granularity of the software clock, then the timer value is rounded up. This means that if, for example, we specified an interval timer to go off each 19,100 microseconds (i.e., just over 19 milliseconds), then, assuming a jiffy value of 4 milliseconds, we would actually get a timer that expired every 20 milliseconds.

High-resolution timers

On modern Linux kernels, the preceding statement that timer resolution is limited by the frequency of the software clock no longer holds true. Since kernel 2.6.21, Linux optionally supports high-resolution timers. If this support is enabled (via the `CONFIG_HIGH_RES_TIMERS` kernel configuration option), then the accuracy of the various timer and sleep interfaces that we describe in this chapter is no longer constrained by the size of the kernel jiffy. Instead, these calls can be as accurate as the underlying hardware allows. On modern hardware, accuracy down to a microsecond is typical.

> The availability of high-resolution timers can be determined by examining the clock resolution returned by *clock_getres()*, described in Section 23.5.1.

23.3 Setting Timeouts on Blocking Operations

One use of real-time timers is to place an upper limit on the time for which a blocking system call can remain blocked. For example, we may wish to cancel a *read()* from a terminal if the user has not entered a line of input within a certain time. We can do this as follows:

1. Call *sigaction()* to establish a handler for SIGALRM, omitting the SA_RESTART flag, so that system calls are not restarted (refer to Section 21.5).
2. Call *alarm()* or *setitimer()* to establish a timer specifying the upper limit of time for which we wish the system call to block.
3. Make the blocking system call.
4. After the system call returns, call *alarm()* or *setitimer()* once more to disable the timer (in case the system call completed before the timer expired).
5. Check to see whether the blocking system call failed with *errno* set to EINTR (interrupted system call).

Listing 23-2 demonstrates this technique for *read()*, using *alarm()* to establish the timer.

Listing 23-2: Performing a *read()* with timeout

timers/timed_read.c

```
#include <signal.h>
#include "tlpi_hdr.h"

#define BUF_SIZE 200

static void     /* SIGALRM handler: interrupts blocked system call */
handler(int sig)
{
    printf("Caught signal\n");          /* UNSAFE (see Section 21.1.2) */
}

int
main(int argc, char *argv[])
{
    struct sigaction sa;
    char buf[BUF_SIZE];
    ssize_t numRead;
    int savedErrno;

    if (argc > 1 && strcmp(argv[1], "--help") == 0)
        usageErr("%s [num-secs [restart-flag]]\n", argv[0]);

    /* Set up handler for SIGALRM. Allow system calls to be interrupted,
       unless second command-line argument was supplied. */

    sa.sa_flags = (argc > 2) ? SA_RESTART : 0;
    sigemptyset(&sa.sa_mask);
```

```
    sa.sa_handler = handler;
    if (sigaction(SIGALRM, &sa, NULL) == -1)
        errExit("sigaction");

    alarm((argc > 1) ? getInt(argv[1], GN_NONNEG, "num-secs") : 10);

    numRead = read(STDIN_FILENO, buf, BUF_SIZE - 1);

    savedErrno = errno;                 /* In case alarm() changes it */
    alarm(0);                           /* Ensure timer is turned off */
    errno = savedErrno;

    /* Determine result of read() */

    if (numRead == -1) {
        if (errno == EINTR)
            printf("Read timed out\n");
        else
            errMsg("read");
    } else {
        printf("Successful read (%ld bytes): %.*s",
                (long) numRead, (int) numRead, buf);
    }

    exit(EXIT_SUCCESS);
}
```

timers/timed_read.c

Note that there is a theoretical race condition in the program in Listing 23-2. If the timer expires after the call to *alarm()*, but before the *read()* call is started, then the *read()* call won't be interrupted by the signal handler. Since the timeout value used in scenarios like this is normally relatively large (at least several seconds) this is highly unlikely to occur, so that, in practice, this is a viable technique. [Stevens & Rago, 2005] proposes an alternative technique using *longjmp()*. A further alternative when dealing with I/O system calls is to use the timeout feature of the *select()* or *poll()* system calls (Chapter 63), which also have the advantage of allowing us to simultaneously wait for I/O on multiple descriptors.

23.4 Suspending Execution for a Fixed Interval (Sleeping)

Sometimes, we want to suspend execution of a process for a fixed amount of time. While it is possible to do this using a combination of *sigsuspend()* and the timer functions already described, it is easier to use one of the sleep functions instead.

23.4.1 Low-Resolution Sleeping: *sleep()*

The *sleep()* function suspends execution of the calling process for the number of seconds specified in the *seconds* argument or until a signal is caught (thus interrupting the call).

```
#include <unistd.h>

unsigned int sleep(unsigned int seconds);
```

Returns 0 on normal completion, or number of unslept seconds if prematurely terminated

If the sleep completes, *sleep()* returns 0. If the sleep is interrupted by a signal, *sleep()* returns the number of remaining (unslept) seconds. As with timers set by *alarm()* and *setitimer()*, system load may mean that the process is rescheduled only at some (normally short) time after the completion of the *sleep()* call.

SUSv3 leaves possible interactions of *sleep()* with *alarm()* and *setitimer()* unspecified. On Linux, *sleep()* is implemented as a call to *nanosleep()* (Section 23.4.2), with the consequence that there is no interaction between *sleep()* and the timer functions. However, on many implementations, especially older ones, *sleep()* is implemented using *alarm()* and a handler for the SIGALRM signal. For portability, we should avoid mixing the use of *sleep()* with *alarm()* and *setitimer()*.

23.4.2 High-Resolution Sleeping: *nanosleep()*

The *nanosleep()* function performs a similar task to *sleep()*, but provides a number of advantages, including finer resolution when specifying the sleep interval.

```
#define _POSIX_C_SOURCE 199309
#include <time.h>

int nanosleep(const struct timespec *request, struct timespec *remain);
```

Returns 0 on successfully completed sleep, or −1 on error or interrupted sleep

The *request* argument specifies the duration of the sleep interval and is a pointer to a structure of the following form:

```
struct timespec {
    time_t tv_sec;          /* Seconds */
    long   tv_nsec;         /* Nanoseconds */
};
```

The *tv_nsec* field specifies a nanoseconds value. It must be a number in the range 0 to 999,999,999.

A further advantage of *nanosleep()* is that SUSv3 explicitly specifies that it should not be implemented using signals. This means that, unlike the situation with *sleep()*, we can portably mix calls to *nanosleep()* with calls to *alarm()* or *setitimer()*.

Although it is not implemented using signals, *nanosleep()* may still be interrupted by a signal handler. In this case, *nanosleep()* returns −1, with *errno* set to the usual EINTR and, if the argument *remain* is not NULL, the buffer it points to returns the remaining unslept time. If desired, we can use the returned value to restart the system call and complete the sleep. This is demonstrated in Listing 23-3. As command-line arguments,

this program expects seconds and nanosecond values for *nanosleep()*. The program loops repeatedly, executing *nanosleep()* until the total sleep interval is passed. If *nanosleep()* is interrupted by the handler for `SIGINT` (generated by typing *Control-C*), then the call is restarted using the value returned in *remain*. When we run this program, we see the following:

```
$ ./t_nanosleep 10 0                    Sleep for 10 seconds
Type Control-C
Slept for:  1.853428 secs
Remaining:  8.146617000
Type Control-C
Slept for:  4.370860 secs
Remaining:  5.629800000
Type Control-C
Slept for:  6.193325 secs
Remaining:  3.807758000
Slept for: 10.008150 secs
Sleep complete
```

Although *nanosleep()* allows nanosecond precision when specifying the sleep interval, the accuracy of the sleep interval is limited to the granularity of the software clock (Section 10.6). If we specify an interval that is not a multiple of the software clock, then the interval is rounded up.

> As noted earlier, on systems that support high-resolution timers, the accuracy of the sleep interval can be much finer than the granularity of the software clock.

This rounding behavior means that if signals are received at a high rate, then there is a problem with the approach employed in the program in Listing 23-3. The problem is that each restart of *nanosleep()* will be subject to rounding errors, since the returned *remain* time is unlikely to be an exact multiple of the granularity of the software clock. Consequently, each restarted *nanosleep()* will sleep longer than the value returned in *remain* by the previous call. In the case of an extremely high rate of signal delivery (i.e., as or more frequent than the software clock granularity), the process may never be able to complete its sleep. On Linux 2.6, this problem can be avoided by making use of *clock_nanosleep()* with the `TIMER_ABSTIME` option. We describe *clock_nanosleep()* in Section 23.5.4.

> In Linux 2.4 and earlier, there is an eccentricity in the implementation of *nanosleep()*. Suppose that a process performing a *nanosleep()* call is stopped by a signal. When the process is later resumed via delivery of `SIGCONT`, then the *nanosleep()* call fails with the error `EINTR`, as expected. However, if the program subsequently restarts the *nanosleep()* call, then the time that the process has spent in the stopped state is *not* counted against the sleep interval, so that the process will sleep longer than expected. This eccentricity is eliminated in Linux 2.6, where the *nanosleep()* call automatically resumes on delivery of the `SIGCONT` signal, and the time spent in the sleep state is counted against the sleep interval.

Listing 23-3: Using *nanosleep()*

timers/t_nanosleep.c

```
#define _POSIX_C_SOURCE 199309
#include <sys/time.h>
#include <time.h>
#include <signal.h>
#include "tlpi_hdr.h"

static void
sigintHandler(int sig)
{
    return;                     /* Just interrupt nanosleep() */
}

int
main(int argc, char *argv[])
{
    struct timeval start, finish;
    struct timespec request, remain;
    struct sigaction sa;
    int s;

    if (argc != 3 || strcmp(argv[1], "--help") == 0)
        usageErr("%s secs nanosecs\n", argv[0]);

    request.tv_sec = getLong(argv[1], 0, "secs");
    request.tv_nsec = getLong(argv[2], 0, "nanosecs");

    /* Allow SIGINT handler to interrupt nanosleep() */

    sigemptyset(&sa.sa_mask);
    sa.sa_flags = 0;
    sa.sa_handler = sigintHandler;
    if (sigaction(SIGINT, &sa, NULL) == -1)
        errExit("sigaction");

    if (gettimeofday(&start, NULL) == -1)
        errExit("gettimeofday");

    for (;;) {
        s = nanosleep(&request, &remain);
        if (s == -1 && errno != EINTR)
            errExit("nanosleep");

        if (gettimeofday(&finish, NULL) == -1)
            errExit("gettimeofday");
        printf("Slept for: %9.6f secs\n", finish.tv_sec - start.tv_sec +
                        (finish.tv_usec - start.tv_usec) / 1000000.0);

        if (s == 0)
            break;                      /* nanosleep() completed */

        printf("Remaining: %2ld.%09ld\n", (long) remain.tv_sec,
                remain.tv_nsec);
```

```
        request = remain;               /* Next sleep is with remaining time */
    }

    printf("Sleep complete\n");
    exit(EXIT_SUCCESS);
}
```
timers/t_nanosleep.c

23.5 POSIX Clocks

POSIX clocks (originally defined in POSIX.1b) provide an API for accessing clocks that measure time with nanosecond precision. Nanosecond time values are represented using the same *timespec* structure as is used by *nanosleep()* (Section 23.4.2).

On Linux, programs using this API must be compiled with the *–lrt* option, in order to link against the *librt* (realtime) library.

The main system calls in the POSIX clocks API are *clock_gettime()*, which retrieves the current value of a clock; *clock_getres()*, which returns the resolution of a clock; and *clock_settime()*, which updates a clock.

23.5.1 Retrieving the Value of a Clock: *clock_gettime()*

The *clock_gettime()* system call returns the time according to the clock specified in *clockid*.

```
#define _POSIX_C_SOURCE 199309
#include <time.h>

int clock_gettime(clockid_t clockid, struct timespec *tp);
int clock_getres(clockid_t clockid, struct timespec *res);
```
Both return 0 on success, or –1 on error

The time value is returned in the *timespec* structure pointed to by *tp*. Although the *timespec* structure affords nanosecond precision, the granularity of the time value returned by *clock_gettime()* may be coarser than this. The *clock_getres()* system call returns a pointer to a *timespec* structure containing the resolution of the clock specified in *clockid*.

The *clockid_t* data type is a type specified by SUSv3 for representing a clock identifier. The first column of Table 23-1 lists the values that can be specified for *clockid*.

Table 23-1: POSIX.1b clock types

Clock ID	Description
CLOCK_REALTIME	Settable system-wide real-time clock
CLOCK_MONOTONIC	Nonsettable monotonic clock
CLOCK_PROCESS_CPUTIME_ID	Per-process CPU-time clock (since Linux 2.6.12)
CLOCK_THREAD_CPUTIME_ID	Per-thread CPU-time clock (since Linux 2.6.12)

The CLOCK_REALTIME clock is a system-wide clock that measures wall-clock time. By contrast with the CLOCK_MONOTONIC clock, the setting of this clock can be changed.

SUSv3 specifies that the CLOCK_MONOTONIC clock measures time since some "unspecified point in the past" that doesn't change after system startup. This clock is useful for applications that must not be affected by discontinuous changes to the system clock (e.g., a manual change to the system time). On Linux, this clock measures the time since system startup.

The CLOCK_PROCESS_CPUTIME_ID clock measures the user and system CPU time consumed by the calling process. The CLOCK_THREAD_CPUTIME_ID clock performs the analogous task for an individual thread within a process.

All of the clocks in Table 23-1 are specified in SUSv3, but only CLOCK_REALTIME is mandatory and widely supported on UNIX implementations.

> Linux 2.6.28 adds a new clock type, CLOCK_MONOTONIC_RAW, to those listed in Table 23-1. This is a nonsettable clock that is similar to CLOCK_MONOTONIC, but it gives access to a pure hardware-based time that is unaffected by NTP adjustments. This nonstandard clock is intended for use in specialized clock-synchronization applications.
>
> Linux 2.6.32 adds two more new clocks to those listed in Table 23-1: CLOCK_REALTIME_COARSE and CLOCK_MONOTIC_COARSE. These clocks are similar to CLOCK_REALTIME and CLOCK_MONOTONIC, but intended for applications that want to obtain lower-resolution timestamps at minimal cost. These nonstandard clocks don't cause any access to the hardware clock (which can be expensive for some hardware clock sources), and the resolution of the returned value is the jiffy (Section 10.6).

23.5.2 Setting the Value of a Clock: *clock_settime()*

The *clock_settime()* system call sets the clock specified by *clockid* to the time supplied in the buffer pointed to by *tp*.

```
#define _POSIX_C_SOURCE 199309
#include <time.h>

int clock_settime(clockid_t clockid, const struct timespec *tp);
```

Returns 0 on success, or −1 on error

If the time specified by *tp* is not a multiple of the clock resolution as returned by *clock_getres()*, the time is rounded downward.

A privileged (CAP_SYS_TIME) process may set the CLOCK_REALTIME clock. The initial value of this clock is typically the time since the Epoch. None of the other clocks in Table 23-1 are modifiable.

> According to SUSv3, an implementation may allow the CLOCK_PROCESS_CPUTIME_ID and CLOCK_THREAD_CPUTIME_ID clocks to be settable. At the time of writing, these clocks are read-only on Linux.

23.5.3 Obtaining the Clock ID of a Specific Process or Thread

The functions described in this section allow us to obtain the ID of a clock that measures the CPU time consumed by a particular process or thread. We can use the returned clock ID in a call to *clock_gettime()* in order to find out the CPU time consumed by the process or thread.

The *clock_getcpuclockid()* function returns the identifier of the CPU-time clock of the process whose ID is *pid*, in the buffer pointed to by *clockid*.

```
#define _XOPEN_SOURCE 600
#include <time.h>

int clock_getcpuclockid(pid_t pid, clockid_t *clockid);
```

Returns 0 on success, or a positive error number on error

If *pid* is 0, *clock_getcpuclockid()* returns the ID of the CPU-time clock of the calling process.

The *pthread_getcpuclockid()* function is the POSIX threads analog of the *clock_getcpuclockid()* function. It returns the identifier of the clock measuring the CPU time consumed by a specific thread of the calling process.

```
#define _XOPEN_SOURCE 600
#include <pthread.h>
#include <time.h>

int pthread_getcpuclockid(pthread_t thread, clockid_t *clockid);
```

Returns 0 on success, or a positive error number on error

The *thread* argument is a POSIX thread ID that identifies the thread whose CPU-time clock ID we want to obtain. The clock ID is returned in the buffer pointed to by *clockid*.

23.5.4 Improved High-Resolution Sleeping: *clock_nanosleep()*

Like *nanosleep()*, the Linux-specific *clock_nanosleep()* system call suspends the calling process until either a specified interval of time has passed or a signal arrives. In this section, we describe the features that distinguish *clock_nanosleep()* from *nanosleep()*.

```
#define _XOPEN_SOURCE 600
#include <time.h>

int clock_nanosleep(clockid_t clockid, int flags,
       const struct timespec *request, struct timespec *remain);
```

Returns 0 on successfully completed sleep,
or a positive error number on error or interrupted sleep

The *request* and *remain* arguments serve similar purposes to the analogous arguments for *nanosleep()*.

By default (i.e., if *flags* is 0), the sleep interval specified in *request* is relative (like *nanosleep()*). However, if we specify TIMER_ABSTIME in *flags* (see the example in Listing 23-4), then *request* specifies an absolute time as measured by the clock identified by *clockid*. This feature is essential in applications that need to sleep accurately until a specific time. If we instead try retrieving the current time, calculating the difference until the desired target time, and doing a relative sleep, then there is a possibility that the process may be preempted in the middle of these steps, and consequently sleep for longer than desired.

As described in Section 23.4.2, this "oversleeping" problem is particularly marked for a process that uses a loop to restart a sleep that is interrupted by a signal handler. If signals are delivered at a high rate, then a relative sleep (of the type performed by *nanosleep()*) can lead to large inaccuracies in the time a process spends sleeping. We can avoid the oversleeping problem by making an initial call to *clock_gettime()* to retrieve the time, adding the desired amount to that time, and then calling *clock_nanosleep()* with the TIMER_ABSTIME flag (and restarting the system call if it is interrupted by a signal handler).

When the TIMER_ABSTIME flag is specified, the *remain* argument is unused (it is unnecessary). If the *clock_nanosleep()* call is interrupted by a signal handler, then the sleep can be restarted by repeating the call with the same *request* argument.

Another feature that distinguishes *clock_nanosleep()* from *nanosleep()* is that we can choose the clock that is used to measure the sleep interval. We specify the desired clock in *clockid*: CLOCK_REALTIME, CLOCK_MONOTONIC, or CLOCK_PROCESS_CPUTIME_ID. See Table 23-1 for a description of these clocks.

Listing 23-4 demonstrates the use of *clock_nanosleep()* to sleep for 20 seconds against the CLOCK_REALTIME clock using an absolute time value.

Listing 23-4: Using *clock_nanosleep()*

```
struct timespec request;

/* Retrieve current value of CLOCK_REALTIME clock */

if (clock_gettime(CLOCK_REALTIME, &request) == -1)
    errExit("clock_gettime");

request.tv_sec += 20;               /* Sleep for 20 seconds from now */

s = clock_nanosleep(CLOCK_REALTIME, TIMER_ABSTIME, &request, NULL);
if (s != 0) {
    if (s == EINTR)
        printf("Interrupted by signal handler\n");
    else
        errExitEN(s, "clock_nanosleep");
}
```

23.6 POSIX Interval Timers

The classical UNIX interval timers set by *setitimer()* suffer a number of limitations:

- We can set only one timer of each of the three types, `ITIMER_REAL`, `ITIMER_VIRTUAL`, and `ITIMER_PROF`.
- The only way of being notified of timer expiration is via delivery of a signal. Furthermore, we can't change the signal that is generated when the timer expires.
- If an interval timer expires multiple times while the corresponding signal is blocked, then the signal handler is called only once. In other words, we have no way of knowing whether there was a *timer overrun*.
- Timers are limited to microsecond resolution. However, some systems have hardware clocks that provide finer resolution than this, and, on such systems, it is desirable to have software access to this greater resolution.

POSIX.1b defined an API to address these limitations, and this API is implemented in Linux 2.6.

> On older Linux systems, an incomplete version of this API was provided via a threads-based implementation in *glibc*. However, this user-space implementation doesn't provide all of the features described here.

The POSIX timer API divides the life of a timer into the following steps:

- The *timer_create()* system call creates a new timer and defines the method by which it will notify the process when it expires.
- The *timer_settime()* system call arms (starts) or disarms (stops) a timer.
- The *timer_delete()* system call deletes a timer that is no longer required.

POSIX timers are not inherited by a child created by *fork()*. They are disarmed and deleted during an *exec()* or on process termination.

On Linux, programs using the POSIX timer API must be compiled with the *-lrt* option, in order to link against the *librt* (realtime) library.

23.6.1 Creating a Timer: *timer_create()*

The *timer_create()* function creates a new timer that measures time using the clock specified by *clockid*.

```
#define _POSIX_C_SOURCE 199309
#include <signal.h>
#include <time.h>

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

Returns 0 on success, or −1 on error

The *clockid* can specify any of the values shown in Table 23-1, or the *clockid* value returned by *clock_getcpuclockid()* or *pthread_getcpuclockid()*. The *timerid* argument points to a buffer that returns a handle used to refer to the timer in later system calls. This buffer is typed as *timer_t*, which is a data type specified by SUSv3 for representing a timer identifier.

The *evp* argument determines how the program is to be notified when the timer expires. It points to a structure of type *sigevent*, defined as follows:

```
union sigval {
    int   sival_int;              /* Integer value for accompanying data */
    void *sival_ptr;              /* Pointer value for accompanying data */
};

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 */
    union {
        pid_t      _tid;          /* ID of thread to be signaled /
        struct {
            void (*_function) (union sigval);
                                  /* Thread notification function */
            void  *_attribute;    /* Really 'pthread_attr_t *' */
        } _sigev_thread;
    } _sigev_un;
};

#define sigev_notify_function    _sigev_un._sigev_thread._function
#define sigev_notify_attributes  _sigev_un._sigev_thread._attribute
#define sigev_notify_thread_id   _sigev_un._tid
```

The *sigev_notify* field of this structure is set to one of the values shown in Table 23-2.

Table 23-2: Values for the *sigev_notify* field of the *sigevent* structure

***sigev_notify* value**	**Notification method**	**SUSv3**
SIGEV_NONE	No notification; monitor timer using *timer_gettime()*	•
SIGEV_SIGNAL	Send signal *sigev_signo* to process	•
SIGEV_THREAD	Call *sigev_notify_function* as start function of new thread	•
SIGEV_THREAD_ID	Send signal *sigev_signo* to thread *sigev_notify_thread_id*	

Further details on the *sigev_notify* field constants, and the fields in the *sigval* structure that are associated with each constant value, are as follows:

SIGEV_NONE
: Don't provide notification of timer expiration. The process can still monitor the progress of the timer using *timer_gettime()*.

SIGEV_SIGNAL
: When the timer expires, generate the signal specified in the *sigev_signo* field for the process. If *sigev_signo* is a realtime signal, then the *sigev_value* field specifies data (an integer or a pointer) to accompany the signal (Section 22.8.1). This data can be retrieved via the *si_value* field of the *siginfo_t* structure that is passed to the handler for this signal or returned by a call to *sigwaitinfo()* or *sigtimedwait()*.

SIGEV_THREAD
: When the timer expires, call the function specified in the *sigev_notify_function* field. This function is invoked *as if* it were the start function in a new thread. The "as if" wording is from SUSv3, and allows an implementation to generate the notifications for a periodic timer either by having each notification delivered to a new unique thread or by having the notifications delivered in series to a single new thread. The *sigev_notify_attributes* field can be specified as NULL or as a pointer to a *pthread_attr_t* structure that defines attributes for the thread (Section 29.8). The union *sigval* value specified in *sigev_value* is passed as the sole argument of the function.

SIGEV_THREAD_ID
: This is similar to SIGEV_SIGNAL, but the signal is sent to the thread whose thread ID matches *sigev_notify_thread_id*. This thread must be in the same process as the calling thread. (With SIGEV_SIGNAL notification, a signal is queued to the process as a whole, and, if there are multiple threads in the process, the signal will be delivered to an arbitrarily selected thread in the process.) The *sigev_notify_thread_id* field can be set to the value returned by *clone()* or the value returned by *gettid()*. The SIGEV_THREAD_ID flag is intended for use by threading libraries. (It requires a threading implementation that employs the CLONE_THREAD option, described in Section 28.2.1. The modern NPTL threading implementation employs CLONE_THREAD, but the older LinuxThreads threading implementation does not.)

All of the above constants are specified in SUSv3, except for SIGEV_THREAD_ID, which is Linux-specific.

The *evp* argument may be specified as NULL, which is equivalent to specifying *sigev_notify* as SIGEV_SIGNAL, *sigev_signo* as SIGALRM (this may be different on other systems, since SUSv3 merely says "a default signal number"), and *sigev_value.sival_int* as the timer ID.

As currently implemented, the kernel preallocates one queued realtime signal structure for each POSIX timer that is created using *timer_create()*. The intent of this preallocation is to ensure that at least one such structure is available for queuing a signal when the timer expires. This means that the number of POSIX timers that may be created is subject to the limitations on the number of realtime signals that can be queued (refer to Section 22.8).

23.6.2 Arming and Disarming a Timer: *timer_settime()*

Once we have created a timer, we can arm (start) or disarm (stop) it using *timer_settime()*.

```
#define _POSIX_C_SOURCE 199309
#include <time.h>

int timer_settime(timer_t timerid, int flags, const struct itimerspec *value,
                  struct itimerspec *old_value);
```

Returns 0 on success, or −1 on error

The *timerid* argument of *timer_settime()* is a timer handle returned by a previous call to *timer_create()*.

The *value* and *old_value* arguments are analogous to the *setitimer()* arguments of the same name: *value* specifies the new settings for the timer, and *old_value* is used to return the previous timer settings (see the description of *timer_gettime()* below). If we are not interested in the previous settings, we can specify *old_value* as NULL. The *value* and *old_value* arguments are pointers to *itimerspec* structures, defined as follows:

```
struct itimerspec {
    struct timespec it_interval;    /* Interval for periodic timer */
    struct timespec it_value;       /* First expiration */
};
```

Each of the fields of the *itimerspec* structure is in turn a structure of type *timespec*, which specifies time values as a number of seconds and nanoseconds:

```
struct timespec {
    time_t tv_sec;                  /* Seconds */
    long   tv_nsec;                 /* Nanoseconds */
};
```

The *it_value* field specifies when the timer will first expire. If either subfield of *it_interval* is nonzero, then this is a periodic timer that, after the initial expiry specified by *it_value*, will expire with the frequency specified in these subfields. If both subfields of *it_interval* are 0, this timer expires just once.

If *flags* is specified as 0, then *value.it_value* is interpreted relative to the clock value at the time of the call to *timer_settime()* (i.e., like *setitimer()*). If *flags* is specified as TIMER_ABSTIME, then *value.it_value* is interpreted as an absolute time (i.e., measured from the clock's zero point). If that time has already passed on the clock, the timer expires immediately.

To arm a timer, we make a call to *timer_settime()* in which either or both of the subfields of *value.it_value* are nonzero. If the timer was previously armed, *timer_settime()* replaces the previous settings.

If the timer value and interval are not multiples of the resolution of the corresponding clock (as returned by *clock_getres()*), these values are rounded up to the next multiple of the resolution.

On each expiration of the timer, the process is notified using the method defined in the *timer_create()* call that created this timer. If the *it_interval* structure contains nonzero values, these values are used to reload the *it_value* structure.

To disarm a timer, we make a call to *timer_settime()* specifying both fields of *value.it_value* as 0.

23.6.3 Retrieving the Current Value of a Timer: *timer_gettime()*

The *timer_gettime()* system call returns the interval and remaining time for the POSIX timer identified by *timerid*.

```
#define _POSIX_C_SOURCE 199309
#include <time.h>

int timer_gettime(timer_t timerid, struct itimerspec *curr_value);
                                                Returns 0 on success, or –1 on error
```

The interval and the time until the next expiration of the timer are returned in the *itimerspec* structure pointed to by *curr_value*. The *curr_value.it_value* field returns the time until next timer expiration, even if this timer was established as an absolute timer using `TIMER_ABSTIME`.

If both fields of the returned *curr_value.it_value* structure are 0, then the timer is currently disarmed. If both fields of the returned *curr_value.it_interval* structure are 0, then the timer expires just once, at the time given in *curr_value.it_value*.

23.6.4 Deleting a Timer: *timer_delete()*

Each POSIX timer consumes a small amount of system resources. Therefore, when we have finished using a timer, we should free these resources by using *timer_delete()* to remove the timer.

```
#define _POSIX_C_SOURCE 199309
#include <time.h>

int timer_delete(timer_t timerid);
                                                Returns 0 on success, or –1 on error
```

The *timerid* argument is a handle returned by a previous call to *timer_create()*. If the timer was armed, then it is automatically disarmed before removal. If there is already a pending signal from an expiration of this timer, that signal remains pending. (SUSv3 leaves this point unspecified, so other UNIX implementations may behave differently.) Timers are deleted automatically when a process terminates.

23.6.5 Notification via a Signal

If we elect to receive timer notifications via a signal, then we can accept the signal via a signal handler, or by calling *sigwaitinfo()* or *sigtimedwait()*. Both mechanisms allow the receiving process to obtain a *siginfo_t* structure (Section 21.4) that provides

further information about the signal. (To take advantage of this feature in a signal handler, we specify the SA_SIGINFO flag when establishing the handler.) The following fields are set in the *siginfo_t* structure:

- *si_signo*: This field contains the signal generated by this timer.
- *si_code*: This field is set to SI_TIMER, indicating that this signal was generated because of the expiration of a POSIX timer.
- *si_value*: This field is set to the value that was supplied in *evp.sigev_value* when the timer was created using *timer_create()*. Specifying different *evp.sigev_value* values provides a means of distinguishing expirations of multiple timers that deliver the same signal.

When calling *timer_create()*, *evp.sigev_value.sival_ptr* is typically assigned the address of the *timerid* argument given in the same call (see Listing 23-5). This allows the signal handler (or the *sigwaitinfo()* call) to obtain the ID of the timer that generated the signal. (Alternatively, *evp.sigev_value.sival_ptr* may be assigned the address of a structure that contains the *timerid* given to *timer_create()*.)

Linux also supplies the following nonstandard field in the *siginfo_t* structure:

- *si_overrun*: This field contains the overrun count for this timer (described in Section 23.6.6).

> Linux also supplies another nonstandard field: *si_timerid*. This field contains an identifier that is used internally by the system to identify the timer (it is not the same as the ID returned by *timer_create()*). It is not useful to applications.

Listing 23-5 demonstrates the use of signals as the notification mechanism for a POSIX timer.

Listing 23-5: POSIX timer notification using a signal

timers/ptmr_sigev_signal.c

```
#define _POSIX_C_SOURCE 199309
#include <signal.h>
#include <time.h>
#include "curr_time.h"                  /* Declares currTime() */
#include "itimerspec_from_str.h"        /* Declares itimerspecFromStr() */
#include "tlpi_hdr.h"

#define TIMER_SIG SIGRTMAX              /* Our timer notification signal */

static void
① handler(int sig, siginfo_t *si, void *uc)
{
    timer_t *tidptr;

    tidptr = si->si_value.sival_ptr;

    /* UNSAFE: This handler uses non-async-signal-safe functions
       (printf(); see Section 21.1.2) */
```

```
    printf("[%s] Got signal %d\n", currTime("%T"), sig);
    printf("    *sival_ptr         = %ld\n", (long) *tidptr);
    printf("    timer_getoverrun() = %d\n", timer_getoverrun(*tidptr));
}

int
main(int argc, char *argv[])
{
    struct itimerspec ts;
    struct sigaction  sa;
    struct sigevent   sev;
    timer_t *tidlist;
    int j;

    if (argc < 2)
        usageErr("%s secs[/nsecs][:int-secs[/int-nsecs]]...\n", argv[0]);

    tidlist = calloc(argc - 1, sizeof(timer_t));
    if (tidlist == NULL)
        errExit("malloc");

    /* Establish handler for notification signal */

    sa.sa_flags = SA_SIGINFO;
    sa.sa_sigaction = handler;
    sigemptyset(&sa.sa_mask);
②   if (sigaction(TIMER_SIG, &sa, NULL) == -1)
        errExit("sigaction");

    /* Create and start one timer for each command-line argument */

    sev.sigev_notify = SIGEV_SIGNAL;    /* Notify via signal */
    sev.sigev_signo = TIMER_SIG;        /* Notify using this signal */

    for (j = 0; j < argc - 1; j++) {
③       itimerspecFromStr(argv[j + 1], &ts);

        sev.sigev_value.sival_ptr = &tidlist[j];
                /* Allows handler to get ID of this timer */

④       if (timer_create(CLOCK_REALTIME, &sev, &tidlist[j]) == -1)
            errExit("timer_create");
        printf("Timer ID: %ld (%s)\n", (long) tidlist[j], argv[j + 1]);

⑤       if (timer_settime(tidlist[j], 0, &ts, NULL) == -1)
            errExit("timer_settime");
    }

⑥   for (;;)                           /* Wait for incoming timer signals */
        pause();
}
```

timers/ptmr_sigev_signal.c

Each of the command-line arguments of the program in Listing 23-5 specifies the initial value and interval for a timer. The syntax of these arguments is described in the program's "usage" message and demonstrated in the shell session below. This program performs the following steps:

- Establish a handler for the signal that is used for timer notifications ②.
- For each command-line argument, create ④ and arm ⑤ a POSIX timer that uses the `SIGEV_SIGNAL` notification mechanism. The *itimerspecFromStr()* function that we use to convert ③ the command-line arguments to *itimerspec* structures is shown in Listing 23-6.
- On each timer expiration, the signal specified in *sev.sigev_signo* will be delivered to the process. The handler for this signal displays the value that was supplied in *sev.sigev_value.sival_ptr* (i.e., the timer ID, *tidlist[j]*) and the overrun value for the timer ①.
- Having created and armed the timers, wait for timer expirations by executing a loop that repeatedly calls *pause()* ⑥.

Listing 23-6 shows the function that converts each of the command-line arguments for the program in Listing 23-5 into a corresponding *itimerspec* structure. The format of the string arguments interpreted by this function is shown in a comment at the top of the listing (and demonstrated in the shell session below).

Listing 23-6: Converting time-plus-interval string to an *itimerspec* value

timers/itimerspec_from_str.c

```
#define _POSIX_C_SOURCE 199309
#include <string.h>
#include <stdlib.h>
#include "itimerspec_from_str.h"        /* Declares function defined here */

/* Convert a string of the following form to an itimerspec structure:
   "value.sec[/value.nanosec][:interval.sec[/interval.nanosec]]".
   Optional components that are omitted cause 0 to be assigned to the
   corresponding structure fields. */

void
itimerspecFromStr(char *str, struct itimerspec *tsp)
{
    char *cptr, *sptr;

    cptr = strchr(str, ':');
    if (cptr != NULL)
        *cptr = '\0';

    sptr = strchr(str, '/');
    if (sptr != NULL)
        *sptr = '\0';

    tsp->it_value.tv_sec = atoi(str);
    tsp->it_value.tv_nsec = (sptr != NULL) ? atoi(sptr + 1) : 0;
```

```
    if (cptr == NULL) {
        tsp->it_interval.tv_sec = 0;
        tsp->it_interval.tv_nsec = 0;
    } else {
        sptr = strchr(cptr + 1, '/');
        if (sptr != NULL)
            *sptr = '\0';
        tsp->it_interval.tv_sec = atoi(cptr + 1);
        tsp->it_interval.tv_nsec = (sptr != NULL) ? atoi(sptr + 1) : 0;
    }
}
```

timers/itimerspec_from_str.c

We demonstrate the use of the program in Listing 23-5 in the following shell session, creating a single timer with an initial timer expiry of 2 seconds and an interval of 5 seconds.

```
$ ./ptmr_sigev_signal 2:5
Timer ID: 134524952 (2:5)
[15:54:56] Got signal 64                    SIGRTMAX is signal 64 on this system
    *sival_ptr         = 134524952          sival_ptr points to the variable tid
    timer_getoverrun() = 0
[15:55:01] Got signal 64
    *sival_ptr         = 134524952
    timer_getoverrun() = 0
Type Control-Z to suspend the process
[1]+  Stopped       ./ptmr_sigev_signal 2:5
```

After suspending the program, we pause for a few seconds, allowing several timer expirations to occur before we resume the program:

```
$ fg
./ptmr_sigev_signal 2:5
[15:55:34] Got signal 64
    *sival_ptr         = 134524952
    timer_getoverrun() = 5
Type Control-C to kill the program
```

The last line of program output shows that five timer overruns occurred, meaning that six timer expirations occurred since the previous signal delivery.

23.6.6 Timer Overruns

Suppose that we have chosen to receive notification of timer expiration via delivery of a signal (i.e., *sigev_notify* is SIGEV_SIGNAL). Suppose further that the timer expires multiple times before the associated signal is caught or accepted. This could occur as the result of a delay before the process is next scheduled. Alternatively, it could occur because delivery of the associated signal was blocked, either explicitly via *sigprocmask()*, or implicitly during the execution of the handler for the signal. How do we know that such *timer overruns* have happened?

We might suppose that using a realtime signal would help solve this problem, since multiple instances of a realtime signal are queued. However, this approach turns out to be unworkable, because there are limits on the number of realtime signals

that can be queued. Therefore, the POSIX.1b committee decided on a different approach: if we choose to receive timer notification via a signal, then multiple instances of the signal are never queued, even if we use a realtime signal. Instead, after receiving the signal (either via a signal handler or by using *sigwaitinfo()*), we can fetch the *timer overrun count*, which is the number of extra timer expirations that occurred between the time the signal was generated and the time it was received. For example, if the timer has expired three times since the last signal was received, then the overrun count is 2.

After receiving a timer signal, we can obtain the timer overrun count in two ways:

- Call *timer_getoverrun()*, which we describe below. This is the SUSv3-specified way of obtaining the overrun count.
- Use the value in the *si_overrun* field of the *siginfo_t* structure returned with the signal. This approach saves the overhead of the *timer_getoverrun()* system call, but is a nonportable Linux extension.

The timer overrun count is reset each time we receive the timer signal. If the timer expired just once since the timer signal was handled or accepted, then the overrun count will be 0 (i.e., there were no overruns).

```
#define _POSIX_C_SOURCE 199309
#include <time.h>

int timer_getoverrun(timer_t timerid);
```

Returns timer overrun count on success, or −1 on error

The *timer_getoverrun()* function returns the overrun count for the timer specified by its *timerid* argument.

The *timer_getoverrun()* function is one of those specified as being async-signal-safe in SUSv3 (Table 21-1, on page 426), so it is safe to call it from within a signal handler.

23.6.7 Notification via a Thread

The SIGEV_THREAD flag allows a program to obtain notification of timer expiration via the invocation of a function in a separate thread. Understanding this flag requires knowledge of POSIX threads that we present later, in Chapters 29 and 30. Readers unfamiliar with POSIX threads may want to read those chapters before examining the example program that we present in this section.

Listing 23-7 demonstrates the use of SIGEV_THREAD. This program takes the same command-line arguments as the program in Listing 23-5. The program performs the following steps:

- For each command-line argument, the program creates ⑥ and arms ⑦ a POSIX timer that uses the SIGEV_THREAD notification mechanism ③.
- Each time this timer expires, the function specified by *sev.sigev_notify_function* ④ will be invoked in a separate thread. When this function is invoked, it receives the value specified in *sev.sigev_value.sival_ptr* as an argument. We assign the

address of the timer ID (*tidlist[jj]*) to this field ⑤ so that the notification function can obtain the ID of the timer that caused its invocation.

- Having created and armed all of the timers, the main program enters a loop that waits for timer expirations ⑧. Each time through the loop, the program uses *pthread_cond_wait()* to wait for a condition variable (*cond*) to be signaled by the thread that is handling a timer notification.
- The *threadFunc()* function is invoked on each timer expiration ①. After printing a message, it increments the value of the global variable *expireCnt*. To allow for the possibility of timer overruns, the value returned by *timer_getoverrun()* is also added to *expireCnt*. (We explained timer overruns in Section 23.6.6 in relation to the `SIGEV_SIGNAL` notification mechanism. Timer overruns can also come into play with the `SIGEV_THREAD` mechanism, because a timer might expire multiple times before the notification function is invoked.) The notification function also signals the condition variable *cond* so that the main program knows to check that a timer has expired ②.

The following shell session log demonstrates the use of the program in Listing 23-7. In this example, the program creates two timers: one with an initial expiry of 5 seconds and an interval of 5 seconds, and the other with an initial expiration of 10 seconds and an interval of 10 seconds.

```
$ ./ptmr_sigev_thread 5:5 10:10
Timer ID: 134525024 (5:5)
Timer ID: 134525080 (10:10)
[13:06:22] Thread notify
    timer ID=134525024
    timer_getoverrun()=0
main(): count = 1
[13:06:27] Thread notify
    timer ID=134525080
    timer_getoverrun()=0
main(): count = 2
[13:06:27] Thread notify
    timer ID=134525024
    timer_getoverrun()=0
main(): count = 3
Type Control-Z to suspend the program
[1]+  Stopped       ./ptmr_sigev_thread 5:5 10:10
$ fg                                          Resume execution
./ptmr_sigev_thread 5:5 10:10
[13:06:45] Thread notify
    timer ID=134525024
    timer_getoverrun()=2                      There were timer overruns
main(): count = 6
[13:06:45] Thread notify
    timer ID=134525080
    timer_getoverrun()=0
main(): count = 7
Type Control-C to kill the program
```

Listing 23-7: POSIX timer notification using a thread function

timers/ptmr_sigev_thread.c

```
#include <signal.h>
#include <time.h>
#include <pthread.h>
#include "curr_time.h"              /* Declaration of currTime() */
#include "tlpi_hdr.h"
#include "itimerspec_from_str.h"    /* Declares itimerspecFromStr() */

static pthread_mutex_t mtx = PTHREAD_MUTEX_INITIALIZER;
static pthread_cond_t cond = PTHREAD_COND_INITIALIZER;

static int expireCnt = 0;           /* Number of expirations of all timers */

static void                         /* Thread notification function */
① threadFunc(union sigval sv)
{
    timer_t *tidptr;
    int s;

    tidptr = sv.sival_ptr;

    printf("[%s] Thread notify\n", currTime("%T"));
    printf("    timer ID=%ld\n", (long) *tidptr);
    printf("    timer_getoverrun()=%d\n", timer_getoverrun(*tidptr));

    /* Increment counter variable shared with main thread and signal
       condition variable to notify main thread of the change. */

    s = pthread_mutex_lock(&mtx);
    if (s != 0)
        errExitEN(s, "pthread_mutex_lock");

    expireCnt += 1 + timer_getoverrun(*tidptr);

    s = pthread_mutex_unlock(&mtx);
    if (s != 0)
        errExitEN(s, "pthread_mutex_unlock");

②   s = pthread_cond_signal(&cond);
    if (s != 0)
        errExitEN(s, "pthread_cond_signal");
}

int
main(int argc, char *argv[])
{
    struct sigevent sev;
    struct itimerspec ts;
    timer_t *tidlist;
    int s, j;
```

```
    if (argc < 2)
        usageErr("%s secs[/nsecs][:int-secs[/int-nsecs]]...\n", argv[0]);

    tidlist = calloc(argc - 1, sizeof(timer_t));
    if (tidlist == NULL)
        errExit("malloc");

③   sev.sigev_notify = SIGEV_THREAD;            /* Notify via thread */
④   sev.sigev_notify_function = threadFunc;     /* Thread start function */
    sev.sigev_notify_attributes = NULL;
            /* Could be pointer to pthread_attr_t structure */

    /* Create and start one timer for each command-line argument */

    for (j = 0; j < argc - 1; j++) {
        itimerspecFromStr(argv[j + 1], &ts);

⑤       sev.sigev_value.sival_ptr = &tidlist[j];
                /* Passed as argument to threadFunc() */

⑥       if (timer_create(CLOCK_REALTIME, &sev, &tidlist[j]) == -1)
            errExit("timer_create");
        printf("Timer ID: %ld (%s)\n", (long) tidlist[j], argv[j + 1]);

⑦       if (timer_settime(tidlist[j], 0, &ts, NULL) == -1)
            errExit("timer_settime");
    }

    /* The main thread waits on a condition variable that is signaled
       on each invocation of the thread notification function. We
       print a message so that the user can see that this occurred. */

    s = pthread_mutex_lock(&mtx);
    if (s != 0)
        errExitEN(s, "pthread_mutex_lock");

⑧   for (;;) {
        s = pthread_cond_wait(&cond, &mtx);
        if (s != 0)
            errExitEN(s, "pthread_cond_wait");
        printf("main(): expireCnt = %d\n", expireCnt);
    }
}
```

timers/ptmr_sigev_thread.c

23.7 Timers That Notify via File Descriptors: the *timerfd* API

Starting with kernel 2.6.25, Linux provides another API for creating timers. The Linux-specific *timerfd* API creates a timer whose expiration notifications can be read from a file descriptor. This is useful because the file descriptor can be monitored along with other descriptors using *select()*, *poll()*, and *epoll* (described in Chapter 63). (With the other timer APIs discussed in this chapter, it requires some effort

to be able to simultaneously monitor one or more timers along with a set of file descriptors.)

The operation of the three new system calls in this API is analogous to the operation of the *timer_create()*, *timer_settime()*, and *timer_gettime()* system calls described in Section 23.6.

The first of the new system calls is *timerfd_create()*, which creates a new timer object and returns a file descriptor referring to that object.

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

int timerfd_create(int clockid, int flags);
                                    Returns file descriptor on success, or –1 on error
```

The value of *clockid* can be either CLOCK_REALTIME or CLOCK_MONOTONIC (see Table 23-1).

In the initial implementation of *timerfd_create()*, the *flags* argument was reserved for future use and had to be specified as 0. However, since Linux 2.6.27, two flags are supported:

TFD_CLOEXEC
: Set the close-on-exec flag (FD_CLOEXEC) for the new file descriptor. This flag is useful for the same reasons as the *open()* O_CLOEXEC flag described in Section 4.3.1.

TFD_NONBLOCK
: Set the O_NONBLOCK flag on the underlying open file description, so that future reads will be nonblocking. This saves additional calls to *fcntl()* to achieve the same result.

When we have finished using a timer created by *timerfd_create()*, we should *close()* the associated file descriptor, so that the kernel can free the resources associated with the timer.

The *timerfd_settime()* system call arms (starts) or disarms (stops) the timer referred to by the file descriptor *fd*.

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

int timerfd_settime(int fd, int flags, const struct itimerspec *new_value,
                    struct itimerspec *old_value);
                                    Returns 0 on success, or –1 on error
```

The *new_value* argument specifies the new settings for the timer. The *old_value* argument can be used to return the previous settings of the timer (see the description of *timerfd_gettime()* below for details). If we are not interested in the previous settings, we can specify *old_value* as NULL. Both of these arguments are *itimerspec* structures that are used in the same way as for *timer_settime()* (see Section 23.6.2).

The *flags* argument is similar to the corresponding argument for *timer_settime()*. It may either be 0, meaning that *new_value.it_value* is interpreted relative to the time of the call to *timerfd_settime()*, or it can be TFD_TIMER_ABSTIME, meaning that

new_value.it_value is interpreted as an absolute time (i.e., measured from the clock's zero point).

The *timerfd_gettime()* system call returns the interval and remaining time for the timer identified by the file descriptor *fd*.

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

int timerfd_gettime(int fd, struct itimerspec *curr_value);
                                                    Returns 0 on success, or –1 on error
```

As with *timer_gettime()*, the interval and the time until the next expiration of the timer are returned in the *itimerspec* structure pointed to by *curr_value*. The *curr_value.it_value* field returns the time until the next timer expiration, even if this timer was established as an absolute timer using `TFD_TIMER_ABSTIME`. If both fields of the returned *curr_value.it_value* structure are 0, then the timer is currently disarmed. If both fields of the returned *curr_value.it_interval* structure are 0, then the timer expires just once, at the time given in *curr_value.it_value*.

Interactions of *timerfd* with *fork()* and *exec()*

During a *fork()*, a child process inherits copies of file descriptors created by *timerfd_create()*. These file descriptors refer to the same timer objects as the corresponding descriptors in the parent, and timer expirations can be read in either process.

File descriptors created by *timerfd_create()* are preserved across an *exec()* (unless the descriptors are marked close-on-exec, as described in Section 27.4), and armed timers will continue to generate timer expirations after the *exec()*.

Reading from the *timerfd* file descriptor

Once we have armed a timer with *timerfd_settime()*, we can use *read()* to read information about timer expirations from the associated file descriptor. For this purpose, the buffer given to *read()* must be large enough to hold an unsigned 8-byte integer (*uint64_t*).

If one or more expirations have occurred since the timer settings were last modified using *timerfd_settime()* or the last *read()* was performed, then *read()* returns immediately, and the returned buffer contains the number of expirations that have occurred. If no timer expirations have occurred, then *read()* blocks until the next expiration occurs. It is also possible to use the *fcntl()* `F_SETFL` operation (Section 5.3) to set the `O_NONBLOCK` flag for the file descriptor, so that reads are nonblocking, and will fail with the error `EAGAIN` if no timer expirations have occurred.

As stated earlier, a *timerfd* file descriptor can be monitored using *select()*, *poll()*, and *epoll*. If the timer has expired, then the file descriptor indicates as being readable.

Example program

Listing 23-8 demonstrates the use of the *timerfd* API. This program takes two command-line arguments. The first argument is mandatory, and specifies the initial time and interval for a timer. (This argument is interpreted using the *itimerspecFromStr()* function shown in Listing 23-6.) The second argument, which is optional, specifies

the maximum number of expirations of the timer that the program should wait for before terminating; the default for this argument is 1.

The program creates a timer using *timerfd_create()*, and arms it using *timerfd_settime()*. It then loops, reading expiration notifications from the file descriptor until the specified number of expirations has been reached. After each *read()*, the program displays the time elapsed since the timer was started, the number of expirations read, and the total number of expirations so far.

In the following shell session log, the command-line arguments specify a timer with a 1-second initial value and 1-second interval, and a maximum of 100 expirations.

```
$ ./demo_timerfd 1:1 100
1.000: expirations read: 1; total=1
2.000: expirations read: 1; total=2
3.000: expirations read: 1; total=3
Type Control-Z to suspend program in background for a few seconds
[1]+  Stopped                 ./demo_timerfd 1:1 100
$ fg                                        Resume program in foreground
./demo_timerfd 1:1 100
14.205: expirations read: 11; total=14      Multiple expirations since last read()
15.000: expirations read: 1; total=15
16.000: expirations read: 1; total=16
Type Control-C to terminate the program
```

From the above output, we can see that multiple timer expirations occurred while the program was suspended in the background, and all of these expirations were returned on the first *read()* after the program resumed execution.

Listing 23-8: Using the *timerfd* API

timers/demo_timerfd.c

```
#include <sys/timerfd.h>
#include <time.h>
#include <stdint.h>                     /* Definition of uint64_t */
#include "itimerspec_from_str.h"        /* Declares itimerspecFromStr() */
#include "tlpi_hdr.h"

int
main(int argc, char *argv[])
{
    struct itimerspec ts;
    struct timespec start, now;
    int maxExp, fd, secs, nanosecs;
    uint64_t numExp, totalExp;
    ssize_t s;

    if (argc < 2 || strcmp(argv[1], "--help") == 0)
        usageErr("%s secs[/nsecs][:int-secs[/int-nsecs]] [max-exp]\n", argv[0]);

    itimerspecFromStr(argv[1], &ts);
    maxExp = (argc > 2) ? getInt(argv[2], GN_GT_0, "max-exp") : 1;
```

```
    fd = timerfd_create(CLOCK_REALTIME, 0);
    if (fd == -1)
        errExit("timerfd_create");

    if (timerfd_settime(fd, 0, &ts, NULL) == -1)
        errExit("timerfd_settime");

    if (clock_gettime(CLOCK_MONOTONIC, &start) == -1)
        errExit("clock_gettime");

    for (totalExp = 0; totalExp < maxExp;) {

        /* Read number of expirations on the timer, and then display
           time elapsed since timer was started, followed by number
           of expirations read and total expirations so far. */

        s = read(fd, &numExp, sizeof(uint64_t));
        if (s != sizeof(uint64_t))
            errExit("read");

        totalExp += numExp;

        if (clock_gettime(CLOCK_MONOTONIC, &now) == -1)
            errExit("clock_gettime");

        secs = now.tv_sec - start.tv_sec;
        nanosecs = now.tv_nsec - start.tv_nsec;
        if (nanosecs < 0) {
            secs--;
            nanosecs += 1000000000;
        }

        printf("%d.%03d: expirations read: %llu; total=%llu\n",
                secs, (nanosecs + 500000) / 1000000,
                (unsigned long long) numExp, (unsigned long long) totalExp);
    }

    exit(EXIT_SUCCESS);
}
```

timers/demo_timerfd.c

23.8 Summary

A process can use *setitimer()* or *alarm()* to set a timer, so that it receives a signal after the passage of a specified amount of real or process time. One use of timers is to set an upper limit on the time for which a system call can block.

Applications that need to suspend execution for a specified interval of real time can use a variety of sleep functions for this purpose.

Linux 2.6 implements the POSIX.1b extensions that define an API for high-precision clocks and timers. POSIX.1b timers provide a number of advantages over traditional (*setitimer()*) UNIX timers. We can: create multiple timers; choose the signal that is delivered on timer expiration; retrieve the timer overrun count in order to

determine if a timer has expired multiple times since the last expiration notification; and choose to receive timer notifications via execution of a thread function instead of delivery of a signal.

The Linux-specific *timerfd* API provides a set of interfaces for creating timers that is similar to the POSIX timers API, but allows timer notifications to be read via a file descriptor. This file descriptor can be monitored using *select()*, *poll()*, and *epoll*.

Further information

Under the rationale for individual functions, SUSv3 provides useful notes on the (standard) timer and sleep interface described in this chapter. [Gallmeister, 1995] discusses POSIX.1b clocks and timers.

23.9 Exercises

23-1. Although *alarm()* is implemented as a system call within the Linux kernel, this is redundant. Implement *alarm()* using *setitimer()*.

23-2. Try running the program in Listing 23-3 (`t_nanosleep.c`) in the background with a 60-second sleep interval, while using the following command to send as many `SIGINT` signals as possible to the background process:

```
$ while true; do kill -INT pid; done
```

You should observe that the program sleeps rather longer than expected. Replace the use of *nanosleep()* with the use of *clock_gettime()* (use a `CLOCK_REALTIME` clock) and *clock_nanosleep()* with the `TIMER_ABSTIME` flag. (This exercise requires Linux 2.6.) Repeat the test with the modified program and explain the difference.

23-3. Write a program to show that if the *evp* argument to *timer_create()* is specified as `NULL`, then this is equivalent to specifying *evp* as a pointer to a *sigevent* structure with *sigev_notify* set to `SIGEV_SIGNAL`, *sigev_signo* set to `SIGALRM`, and *si_value.sival_int* set to the timer ID.

23-4. Modify the program in Listing 23-5 (`ptmr_sigev_signal.c`) to use *sigwaitinfo()* instead of a signal handler.