CHAPTER 11
Time

Time serves various purposes in a modern operating system, and many programs need to keep track of it. The kernel measures the passage of time in three different ways:

Wall time (or real time)
: This is the actual time and date in the real world—that is, the time as one would read it on a clock on the wall. Processes use the wall time when interfacing with the user or timestamping an event.

Process time
: This is the time that a process spends executing on a processor. It can be a measurement of the time the process itself spent executing (*user time*) or the time the kernel spent working on the process's behalf (*system time*). Processes care about process time for profiling, auditing, and statistical purposes, for example, measuring how much processor time a given algorithm took to complete. Wall time is misleading for such uses because, given the multitasking nature of Linux, wall time is generally greater than process time. Conversely, given multiple processors and a threaded process, the process time can actually exceed the wall time for a given operation!

Monotonic time
: This time source is strictly linearly increasing. Most operating systems, Linux included, use the system's uptime (time since boot) for this purpose. The wall time can change—for example, because the user may set it or because the system continually adjusts the time to combat clock skew—and additional imprecision can be introduced through, say, leap seconds. The system uptime, on the other hand, is a deterministic and unchangeable representation of time. The important aspect of a monotonic time source is not the current value but the guarantee that the time source is strictly linearly increasing and thus useful for calculating the difference in time between two samplings.

These three measurements of time can be represented in one of two formats:

Relative time
: This is a value relative to some benchmark, such as the current instant: for example, *5 seconds from now*, or *10 minutes ago*. Monotonic time is useful for calculating relative time.

Absolute time
: This represents time without any such benchmark: say, *noon on 25 March 1968*. Wall time is ideal for calculating absolute time.

Both relative and absolute forms of time have uses. A process might need to cancel a request in 500 milliseconds, refresh the screen 60 times per second, or note that 7 seconds have elapsed since an operation began. All of these call for relative time calculations. Conversely, a calendar application might save the date for the user's toga party as 8 February, a filesystem will write out the full date and time when a file is created (rather than "5 seconds ago"), and the user's clock displays the Gregorian date, not the number of seconds since the system booted.

Unix systems represent absolute time as the number of elapsed seconds since the *epoch*, which is defined as 00:00:00 UTC on the morning of 1 January 1970. UTC (Coordinated Universal Time) is roughly GMT (Greenwich Mean Time) or Zulu time. Curiously, this means that in Unix, even absolute time is, at a low level, relative. Unix introduces a special data type for storing "seconds since the epoch," which we will look at in the next section.

Operating systems track the progression of time via the *software clock*, a clock maintained by the kernel in software. The kernel instantiates a periodic timer, known as the *system timer*, that pops at a specific frequency. When a timer interval ends, the kernel increments the elapsed time by one unit, known as a *tick* or a *jiffy*. The counter of elapsed ticks is known as the *jiffies counter*. Previously a 32-bit value, jiffies is a 64-bit counter as of the 2.6 Linux kernel.[1]

On Linux, the frequency of the system timer is called `HZ` because a preprocessor define of the selfsame name represents it. The value of `HZ` is architecture-specific and not part of the Linux ABI. Thus, programs cannot depend on or expect any given value. Historically, the x86 architecture used a value of `100`, meaning the system timer ran 100 times per second (that is, the system timer had a frequency of 100 hertz). This gave each jiffy a value of 0.01 seconds. With the release of the 2.6 Linux kernel, the kernel developers bumped the value of `HZ` to `1000`, giving each jiffy a value of 0.001 seconds. However,

1. The Linux kernel now supports "tickless" operation, so this is no longer strictly true.

in version 2.6.13 and later, `HZ` is `250`, providing each jiffy a value of 0.004 seconds.[2] There is a trade-off inherent in the value of `HZ`: higher values provide higher resolution but incur greater timer overhead.

Although processes should not rely on any fixed value of `HZ`, POSIX defines a mechanism for determining the system timer frequency at runtime:

```
long hz;

hz = sysconf (_SC_CLK_TCK);
if (hz == -1)
        perror ("sysconf"); /* should never occur */
```

This interface is useful when a program wants to determine the resolution of the system's timer, but it is not needed for converting system time values to seconds because most POSIX interfaces export measurements of time that are already converted or that are scaled to a fixed frequency, independent of `HZ`. Unlike `HZ`, this fixed frequency is part of the system ABI; on x86, the value is `100`. POSIX functions that return time in terms of clock ticks use `CLOCKS_PER_SEC` to represent the fixed frequency.

Occasionally, events conspire to turn off a computer. Sometimes, computers are even unplugged; yet, upon boot, they have the correct time. This is because most computers have a battery-powered *hardware clock* that stores the time and date while the computer is off. When the kernel boots, it initializes its concept of the current time from the hardware clock. Likewise, when the user shuts down the system, the kernel writes the current time back to the hardware clock. The system's administrator may synchronize time at other points via the *hwclock* command.

Managing the passage of time on a Unix system involves several tasks, only some of which any given process is concerned with: they include setting and retrieving the current wall time, calculating elapsed time, sleeping for a given amount of time, performing high-precision measurements of time, and controlling timers. This chapter covers this full range of time-related chores. We'll begin by looking at the data structures with which Linux represents time.

Time's Data Structures

As Unix systems evolved, implementing their own interfaces for managing time, multiple data structures came to represent the seemingly simple concept of time. These data structures range from the simple integer to various multifield structures. We'll cover them here before we dive into the actual interfaces.

2. `HZ` is also now a compile-time kernel option, with the values `100`, `250`, `300`, and `1000` supported on the x86 architecture. Regardless, user space cannot depend on any particular value for `HZ`.

The Original Representation

The simplest data structure is `time_t`, defined in the header `<time.h>`. The intention was for `time_t` to be an opaque type. However, on most Unix systems, including Linux, the type is a simple typedef to the C `long` type:

```
typedef long time_t;
```

`time_t` represents the number of elapsed seconds since the epoch. "That won't last long before overflowing!" is a typical response. In fact, it will last longer than you might expect, but it indeed will overflow while plenty of Unix systems are still in use. With a 32-bit `long` type, `time_t` can represent up to 2,147,483,647 seconds past the epoch. This suggests that we will have the Y2K mess all over again—in *2038*! With luck, however, come 22:14:07 on Monday, 18 January 2038, most systems and software will be 64-bit.

And Now, Microsecond Precision

Another issue with `time_t` is that a lot can happen in a single second. The `timeval` structure extends `time_t` to add microsecond precision. The header `<sys/time.h>` defines this structure as follows:

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

struct timeval {
        time_t      tv_sec;     /* seconds */
        suseconds_t tv_usec;    /* microseconds */
};
```

`tv_sec` measures seconds, and `tv_usec` measures microseconds. The confusing `suseconds_t` is normally a typedef to an integer type.

Even Better: Nanosecond Precision

Not content with microsecond resolution, the `timespec` structure ups the ante to nanosecond resolution. The header `<time.h>` defines this structure as follows:

```
#include <time.h>

struct timespec {
        time_t  tv_sec;       /* seconds */
        long    tv_nsec;      /* nanoseconds */
};
```

Given the choice, interfaces prefer nanosecond to microsecond resolution In addition, the `timespec` structure dropped the silly `suseconds_t` business in favor of a simple and unpretentious `long`. Consequently, since the introduction of the `timespec` structure, most time-related interfaces have switched to it and thus have gained greater precision. However, as we will see, one important function still uses `timeval`.

In practice, neither structure usually offers the stated precision because the system timer is not providing nanosecond or even microsecond resolution. Nonetheless, it's preferable to have the resolution available in the interface so it can accommodate whatever resolution the system does offer.

Breaking Down Time

Some of the functions that we will cover convert between Unix time and strings or programmatically build a string representing a given date. To facilitate this process, the C standard provides the `tm` structure for representing "broken-down" time in a more human-readable format. This structure is also defined in `<time.h>`:

```
#include <time.h>

struct tm {
        int tm_sec;           /* seconds */
        int tm_min;           /* minutes */
        int tm_hour;          /* hours */
        int tm_mday;          /* the day of the month */
        int tm_mon;           /* the month */
        int tm_year;          /* the year */
        int tm_wday;          /* the day of the week */
        int tm_yday;          /* the day in the year */
        int tm_isdst;         /* daylight savings time? */
#ifdef _BSD_SOURCE
        long tm_gmtoff;       /* time zone's offset from GMT */
        const char *tm_zone;  /* time zone abbreviation */
#endif /* _BSD_SOURCE */
};
```

The `tm` structure makes it easier to tell whether a `time_t` value of, say, `314159` is a Sunday or a Saturday (it is the former). In terms of space, it is obviously a poor choice for representing the date and time, but it is handy for converting to and from user-oriented values.

The fields are as follows:

`tm_sec`
: The number of seconds after the minute. This value normally ranges from `0` to `59`, but it can be as high as `61` to indicate up to two leap seconds.

`tm_min`
: The number of minutes after the hour. This value ranges from `0` to `59`.

`tm_hour`
: The number of hours after midnight. This value ranges from `0` to `23`.

`tm_mday`
: The day of the month. This value ranges from `0` to `31`. POSIX does not specify the value `0`; however, Linux uses it to indicate the last day of the preceding month.

`tm_mon`
: The number of months since January. This value ranges from `0` to `11`.

`tm_year`
: The number of years since 1900.

`tm_wday`
: The number of days since Sunday. This value ranges from `0` to `6`.

`tm_yday`
: The number of days since 1 January. This value ranges from `0` to `365`.

`tm_isdst`
: A special value indicating whether daylight savings time (DST) is in effect at the time described by the other fields. If the value is positive, DST is in effect. If it is `0`, DST is not in effect. If the value is negative, the state of DST is unknown.

`tm_gmtoff`
: The offset in seconds of the current time zone from Greenwich Mean Time. This field is present only if `_BSD_SOURCE` is defined before including `<time.h>`.

`tm_zone`
: The abbreviation for the current time zone—for example, `EST`. This field is present only if `_BSD_SOURCE` is defined before including `<time.h>`.

A Type for Process Time

The type `clock_t` represents clock ticks. It is an integer type, often a `long`. Depending on the interface, the ticks that `clock_t` represent are the system's actual timer frequency (`HZ`) or `CLOCKS_PER_SEC`.

POSIX Clocks

Several of the system calls discussed in this chapter utilize *POSIX clocks*, a standard for implementing and representing time sources. The type `clockid_t` represents a specific POSIX clock, five of which Linux supports:

`CLOCK_REALTIME`
: The system-wide real time (wall time) clock. Setting this clock requires special privileges.

`CLOCK_MONOTONIC`
: A monotonically increasing clock that is not settable by any process. It represents the elapsed time since some unspecified starting point, such as system boot.

`CLOCK_MONOTONIC_RAW`
: Similar to `CLOCK_MONOTONIC`, except the clock is not eligible for slewing (correction for clock skew). That is, if the hardware clock runs faster or slower than wall time, it won't be adjusted when read via this clock. This clock is Linux-specific.

`CLOCK_PROCESS_CPUTIME_ID`
: A high-resolution, per-process clock available from the processor. For example, on the x86 architecture, this clock uses the timestamp counter (TSC) register.

`CLOCK_THREAD_CPUTIME_ID`
: Similar to the per-process clock, but unique to each thread in a process.

POSIX requires only `CLOCK_REALTIME`. Therefore, while Linux reliably provides all five clocks, portable code should rely only on `CLOCK_REALTIME`.

Time Source Resolution

POSIX defines the function `clock_getres()` for obtaining the resolution of a given time source:

```
#include <time.h>

int clock_getres (clockid_t clock_id,
                  struct timespec *res);
```

A successful call to `clock_getres()` stores the resolution of the clock specified by `clock_id` in `res` if it is not `NULL` and returns `0`. On failure, the function returns `-1` and sets `errno` to one of the following two error codes:

`EFAULT`
: `res` is an invalid pointer.

`EINVAL`
: `clock_id` is not a valid time source on this system.

The following example outputs the resolution of the five time sources discussed in the previous section:

```
clockid_t clocks[] = {
        CLOCK_REALTIME,
        CLOCK_MONOTONIC,
        CLOCK_PROCESS_CPUTIME_ID,
        CLOCK_THREAD_CPUTIME_ID,
        CLOCK_MONOTONIC_RAW,
        (clockid_t) -1 };
```

```
int i;


for (i = 0; clocks[i] != (clockid_t) -1; i++) {
        struct timespec res;
        int ret;

        ret = clock_getres (clocks[i], &res);
        if (ret)
                perror ("clock_getres");
        else
                printf ("clock=%d sec=%ld nsec=%ld\n",
                        clocks[i], res.tv_sec, res.tv_nsec);
}
```

On a modern x86 system, the output resembles the following:

```
clock=0 sec=0 nsec=4000250
clock=1 sec=0 nsec=4000250
clock=2 sec=0 nsec=1
clock=3 sec=0 nsec=1
clock=4 sec=0 nsec=4000250
```

Note that 4,000,250 nanoseconds is 4 milliseconds, which is 0.004 seconds. In turn, 0.004 seconds is the resolution of the x86 system clock given an `HZ` value of `250`, as we discussed in the first section of this chapter. Thus, we see that both `CLOCK_REALTIME` and `CLOCK_MONOTONIC` are tied to jiffies and the resolution provided by the system timer. Conversely, both `CLOCK_PROCESS_CPUTIME_ID` and `CLOCK_PROCESS_CPUTIME_ID` utilize a higher-resolution time source—on this x86 machine, the TSC, which we see provides nanosecond resolution.

On Linux (and most other Unix systems), all of the functions that use POSIX clocks require linking the resulting object file with *librt*. For example, if compiling the previous snippet into a complete executable, you might use the following command:

```
$ gcc -Wall -W -O2 -lrt -g -o snippet snippet.c
```

Getting the Current Time of Day

Applications have several reasons for desiring the current time and date: to display it to the user, to calculate relative or elapsed time, to timestamp an event, and so on. The simplest and historically most common way of obtaining the current time is the `time()` function:

```
#include <time.h>

time_t time (time_t *t);
```

A call to `time()` returns the current time represented as the number of seconds elapsed since the epoch. If the parameter `t` is not `NULL`, the function also writes the current time into the provided pointer.

On error, the function returns `-1` (typecast to a `time_t`) and sets `errno` appropriately. The only possible error is `EFAULT`, noting that `t` is an invalid pointer.

For example:

```
time_t t;

printf ("current time: %ld\n", (long) time (&t));
printf ("the same value: %ld\n", (long) t);
```

A Consistent but Inaccurate View of Time

`time_t`'s representation of "seconds elapsed since the epoch" is not the actual number of seconds that have passed since that fateful moment in time. The Unix calculation assumes leap years are all years divisible by four and ignores leap seconds altogether. The point of the `time_t` representation is not that it is accurate, but that it is consistent—and it is.

A Better Interface

The function `gettimeofday()` extends `time()` by offering microsecond resolution:

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

int gettimeofday (struct timeval *tv,
                  struct timezone *tz);
```

A successful call to `gettimeofday()` places the current time in the `timeval` structure pointed at by `tv` and returns `0`. The `timezone` structure and the `tz` parameter are obsolete; neither should be used on Linux. Always pass `NULL` for `tz`.

On failure, the call returns `-1` and sets `errno` to `EFAULT`; this is the only possible error, signifying that `tv` or `tz` is an invalid pointer.

For example:

```
struct timeval tv;
int ret;

ret = gettimeofday (&tv, NULL);
if (ret)
        perror ("gettimeofday");
else
        printf ("seconds=%ld useconds=%ld\n",
                (long) tv.sec, (long) tv.tv_usec);
```

The `timezone` structure is obsolete because the kernel does not manage the time zone, and *glibc* refuses to use the `timezone` structure's `tz_dsttime` field. We will look at manipulating the time zone in a subsequent section.

An Advanced Interface

POSIX provides the `clock_gettime()` interface for obtaining the time of a specific time source. More useful, however, is that the function allows for nanosecond precision:

```
#include <time.h>

int clock_gettime (clockid_t clock_id,
                   struct timespec *ts);
```

On success, the call returns `0` and stores the current time of the time source specified by `clock_id` in `ts`. On failure, the call returns `-1` and sets `errno` to one of the following:

`EFAULT`
: `ts` is an invalid pointer.

`EINVAL`
: `clock_id` is an invalid time source on this system.

The following example obtains the current time of all four of the standard time sources:

```
clockid_t clocks[] = {
        CLOCK_REALTIME,
        CLOCK_MONOTONIC,
        CLOCK_PROCESS_CPUTIME_ID,
        CLOCK_THREAD_CPUTIME_ID,
        CLOCK_MONOTONIC_RAW,
        (clockid_t) -1 };
int i;

for (i = 0; clocks[i] != (clockid_t) -1; i++) {
        struct timespec ts;
        int ret;

        ret = clock_gettime (clocks[i], &ts);
        if (ret)
                perror ("clock_gettime");
        else
                printf ("clock=%d sec=%ld nsec=%ld\n",
                        clocks[i], ts.tv_sec, ts.tv_nsec);
}
```

Getting the Process Time

The `times()` system call retrieves the process time of the running process and its children, in clock ticks:

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

struct tms {
        clock_t tms_utime;   /* user time consumed */
        clock_t tms_stime;   /* system time consumed */
        clock_t tms_cutime;  /* user time consumed by children */
        clock_t tms_cstime;  /* system time consumed by children */
};

clock_t times (struct tms *buf);
```

On success, the call fills the provided `tms` structure pointed at by `buf` with the process time consumed by the invoking process and its children. The reported times are broken into user and system time. *User time* is the time spent executing code in user space. *System time* is the time spent executing code in kernel space—for example, during a system call, or a page fault. The reported times for each child are included only after the child terminates, and the parent invokes `waitpid()` (or a related function) on the process. The call returns the number of clock ticks, monotonically increasing, elapsed since an arbitrary but fixed point in the past. This reference point was once system boot—thus, the `times()` function returned the system uptime in ticks—but the reference point is now about 429 million seconds before system boot. The kernel developers implemented this change to catch kernel code that could not handle the system uptime wrapping around and hitting zero. The absolute value of this function's return is thus worthless; relative changes between two invocations, however, continue to have value.

On failure, the call returns `-1` and sets `errno` as appropriate. On Linux, the only possible error code is `EFAULT`, signifying that `buf` is an invalid pointer.

Setting the Current Time of Day

While previous sections have described how to retrieve times, applications occasionally also need to set the current time and date to a provided value. This is almost always handled by a utility designed solely for this purpose, such as *date*.

The time-setting counterpart to `time()` is `stime()`:

```
#define _SVID_SOURCE
#include <time.h>

int stime (time_t *t);
```

A successful call to `stime()` sets the system time to the value pointed at by `t` and returns `0`. The call requires that the invoking user have the `CAP_SYS_TIME` capability. Generally, only the root user wields this capability.

On failure, the call returns `-1` and sets `errno` to `EFAULT`, signifying that `t` was an invalid pointer, or `EPERM`, signifying that the invoking user did not possess the `CAP_SYS_TIME` capability.

Usage is very simple:

```
time_t t = 1;
int ret;

/* set time to one second after the epoch */
ret = stime (&t);
if (ret)
        perror ("stime");
```

We will look at functions that make it easier to convert human-readable forms of time to a `time_t` in a subsequent section.

Setting Time with Precision

The counterpart to `gettimeofday()` is `settimeofday()`:

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

int settimeofday (const struct timeval *tv,
                  const struct timezone *tz);
```

A successful call to `settimeofday()` sets the system time as given by `tv` and returns `0`. As with `gettimeofday()`, passing `NULL` for `tz` is the best practice. On failure, the call returns `-1` and sets `errno` to one of the following:

`EFAULT`
: `tv` or `tz` points at an invalid region of memory.

`EINVAL`
: A field in one of the provided structures is invalid.

`EPERM`
: The calling process lacks the `CAP_SYS_TIME` capability.

The following example sets the current time to a Saturday in the middle of December 1979:

```
struct timeval tv = { .tv_sec  = 31415926,
                      .tv_usec = 27182818 };
int ret;

ret = settimeofday (&tv, NULL);
if (ret)
        perror ("settimeofday");
```

An Advanced Interface for Setting the Time

Just as `clock_gettime()` improves on `gettimeofday()`, `clock_settime()` obsoleces `settimeofday()`:

```
#include <time.h>

int clock_settime (clockid_t clock_id,
                   const struct timespec *ts);
```

On success, the call returns `0`, and the time source specified by `clock_id` is set to the time specified by `ts`. On failure, the call returns `-1` and sets `errno` to one of the following:

`EFAULT`
: `ts` is an invalid pointer.

`EINVAL`
: `clock_id` is an invalid time source on this system.

`EPERM`
: The process lacks the needed permissions to set the specified time source, or the specified time source may not be set.

On most systems, the only settable time source is `CLOCK_REALTIME`. Thus, the only advantage of this function over `settimeofday()` is that it offers nanosecond precision (along with not having to deal with the worthless `timezone` structure).

Playing with Time

Unix systems and the C language provide a family of functions for converting between broken-down time (an ASCII string representation of time) and `time_t`. `asctime()` converts a `tm` structure—broken-down time—to an ASCII string:

```
#include <time.h>

char * asctime (const struct tm *tm);
char * asctime_r (const struct tm *tm, char *buf);
```

It returns a pointer to a statically allocated string. A subsequent call to any time function may overwrite this string; `asctime()` is not thread-safe.

Thus, multithreaded programs (and developers who loathe poorly designed interfaces) should use `asctime_r()`. Instead of returning a pointer to a statically allocated string, this function uses the string provided via `buf`, which must be at least 26 characters in length.

Both functions return `NULL` in the case of error.

`mktime()` also converts a `tm` structure, but it converts it to a `time_t`:

```
#include <time.h>

time_t mktime (struct tm *tm);
```

`mktime()` also sets the time zone via `tzset()`, as specified by `tm`. On error, it returns `-1` (typecast to a `time_t`).

`ctime()` converts a `time_t` to its ASCII representation:

```
#include <time.h>

char * ctime (const time_t *timep);
char * ctime_r (const time_t *timep, char *buf);
```

On failure, it returns `NULL`. For example:

```
time_t t = time (NULL);

printf ("the time a mere line ago: %s", ctime (&t));
```

Note the lack of newline. Perhaps inconveniently, `ctime()` appends a newline to its returned string.

Like `asctime()`, `ctime()` returns a pointer to a static string. As this is not thread-safe, threaded programs should instead use `ctime_r()`, which operates on the buffer provided by `buf`. The buffer must be at least 26 characters in length.

`gmtime()` converts the given `time_t` to a `tm` structure, expressed in terms of the UTC time zone:

```
#include <time.h>

struct tm * gmtime (const time_t *timep);
struct tm * gmtime_r (const time_t *timep, struct tm *result);
```

On failure, it returns `NULL`.

This function statically allocates the returned structure and is thus thread-unsafe. Threaded programs should use `gmtime_r()`, which operates on the structure pointed at by `result`.

`localtime()` and `localtime_r()` perform functions akin to `gmtime()` and `gmtime_r()`, respectively, but they express the given `time_t` in terms of the user's time zone:

```
#include <time.h>

struct tm * localtime (const time_t *timep);
struct tm * localtime_r (const time_t *timep, struct tm *result);
```

As with `mktime()`, a call to `localtime()` also calls `tzset()` and initializes the time zone. Whether `localtime_r()` performs this step is unspecified.

`difftime()` returns the number of seconds that have elapsed between two `time_t` values, cast to a `double`:

```
#include <time.h>

double difftime (time_t time1, time_t time0);
```

On all POSIX systems, `time_t` is an arithmetic type, and `difftime()` is equivalent to the following, excepting detection of overflow in the subtraction:

```
(double) (time1 - time0)
```

On Linux, because `time_t` is an integer type, there is no need for the cast to `double`. To remain portable, use `difftime()`.

Tuning the System Clock

Large and abrupt jumps in the wall clock time can wreak havoc on applications that depend on absolute time for their operation. Consider as an example *make*, which builds software projects as detailed by a *Makefile*. Each invocation of the program does not rebuild entire source trees; if it did, in large software projects, a single changed file could result in hours of rebuilding. Instead, *make* looks at the file modification timestamps of the source file (say, *wolf.c*) versus the object file (*wolf.o*). If the source file or any of its prerequisites, such as *wolf.h*, is newer than the object file, *make* rebuilds the source file into an updated object file. If the source file is not newer than the object, no action is taken.

With this in mind, consider what might happen if the user realized his clock was off by a couple of hours and ran *date* to update the system clock. If the user then updated and resaved *wolf.c*, we could have trouble. If the user has moved the current time backward, *wolf.c* will look older than *wolf.o*, even though it isn't, and no rebuild will occur.

To prevent such a debacle, Unix provides the `adjtime()` function, which gradually adjusts the current time in the direction of a given delta. The intention is for background activities such as Network Time Protocol (NTP) daemons, which constantly adjust the time in correction of clock skew, to use `adjtime()` to minimize their effects on the system. The periodic adjustment of the clock to correct for clock skew is called *slewing*:

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

int adjtime (const struct timeval *delta,
             struct timeval *olddelta);
```

A successful call to `adjtime()` instructs the kernel to slowly begin adjusting the time as stipulated by `delta`, and then returns `0`. If the time specified by `delta` is positive, the kernel speeds up the system clock by `delta` until the correction is fully applied. If the time specified by `delta` is negative, the kernel slows down the system clock until the correction is applied. The kernel applies all adjustments such that the clock is always monotonically increasing and never undergoes an abrupt time change. Even with a

negative delta, the adjustment will not move the clock backward; instead, the clock slows down until the system time converges with the corrected time.

If delta is not NULL, the kernel stops processing any previously registered corrections. However, the part of the correction already made, if any, is maintained. If olddelta is not NULL, any previously registered and yet unapplied correction is written into the provided timeval structure. Passing a NULL delta and a valid olddelta allows retrieval of any ongoing correction.

The corrections applied by adjtime() should be small. The ideal use case is NTP, which applies only small corrections (but a handful of seconds). Linux maintains minimum and maximum correction thresholds of a few thousand seconds in either direction.

On error, adjtime() returns –1 and sets errno to one of these values:

EFAULT
: delta or olddelta is an invalid pointer.

EINVAL
: The adjustment delineated by delta is too large or too small.

EPERM
: The invoking user does not possess the CAP_SYS_TIME capability.

RFC 1305 defines a significantly more powerful and correspondingly more complex clock-adjustment algorithm than the gradual correction approach undertaken by adjtime(). Linux implements this algorithm with the adjtimex() system call:

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

int adjtimex (struct timex *adj);
```

A call to adjtimex() reads kernel time-related parameters into the timex structure pointed at by adj. Optionally, depending on the modes field of this structure, the system call may additionally set certain parameters.

The header <sys/timex.h> defines the timex structure as follows:

```
struct timex {
        int modes;           /* mode selector */
        long offset;         /* time offset (usec) */
        long freq;           /* frequency offset (scaled ppm) */
        long maxerror;       /* maximum error (usec) */
        long esterror;       /* estimated error (usec) */
        int status;          /* clock status */
        long constant;       /* PLL time constant */
        long precision;      /* clock precision (usec) */
        long tolerance;      /* clock frequency tolerance (ppm) */
        struct timeval time; /* current time */
```

```
        long tick;          /* usecs between clock ticks */
};
```

The `modes` field is a bitwise OR of zero or more of the following flags:

`ADJ_OFFSET`
: Set the time offset via `offset`.

`ADJ_FREQUENCY`
: Set the frequency offset via `freq`.

`ADJ_MAXERROR`
: Set the maximum error via `maxerror`.

`ADJ_ESTERROR`
: Set the estimated error via `esterror`.

`ADJ_STATUS`
: Set the clock status via `status`.

`ADJ_TIMECONST`
: Set the phase-locked loop (PLL) time constant via `constant`.

`ADJ_TICK`
: Set the tick value via `tick`.

`ADJ_OFFSET_SINGLESHOT`
: Set the time offset via `offset` once with a simple algorithm like `adjtime()`.

If `modes` is `0`, no values are set. Only a user with the `CAP_SYS_TIME` capability may provide a nonzero `modes` value; any user may provide `0` for `modes`, retrieving all of the parameters but setting none of them.

On success, `adjtimex()` returns the current clock state, which is one of the following:

`TIME_OK`
: The clock is synchronized.

`TIME_INS`
: A leap second will be inserted.

`TIME_DEL`
: A leap second will be deleted.

`TIME_OOP`
: A leap second is in progress.

`TIME_WAIT`
: A leap second just occurred.

`TIME_BAD`
: The clock is not synchronized.

On failure, `adjtimex()` returns `-1` and sets `errno` to one of the following error codes:

`EFAULT`
: `adj` is an invalid pointer.

`EINVAL`
: One or more of `modes`, `offset`, or `tick` is invalid.

`EPERM`
: `modes` is nonzero, but the invoking user does not possess the `CAP_SYS_TIME` capability.

The `adjtimex()` system call is Linux-specific. Applications concerned with portability should prefer `adjtime()`.

RFC 1305 defines a complex algorithm, so a complete discussion of `adjtimex()` is outside the scope of this book. For more information, see the RFC.

Sleeping and Waiting

Various functions allow a process to sleep (suspend execution) for a given amount of time. The first such function, `sleep()`, puts the invoking process to sleep for the number of seconds specified by `seconds`:

```
#include <unistd.h>

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

The call returns the number of seconds *not* slept. Thus, a successful call returns `0`, but the function may return other values between `0` and `seconds` inclusive (if, say, a signal interrupts the nap). The function does not set `errno`. Most users of `sleep()` do not care about how long the process actually slept and, consequently, do not check the return value:

```
sleep (7);        /* sleep seven seconds */
```

If sleeping the entire specified time is truly a concern, you can continue calling `sleep()` with its return value until it returns `0`:

```
unsigned int s = 5;

/* sleep five seconds: no ifs, ands, or buts about it */
while ((s = sleep (s)))
        ;
```

Sleeping with Microsecond Precision

Sleeping with whole-second granularity is pretty lame. A second is an eternity on a modern system, so programs often want to sleep with subsecond resolution. Enter `usleep()`:

```
/* BSD version */
#include <unistd.h>

void usleep (unsigned long usec);

/* SUSv2 version */
#define _XOPEN_SOURCE 500
#include <unistd.h>

int usleep (useconds_t usec);
```

A successful call to `usleep()` puts the invoking process to sleep for `usec` microseconds. Unfortunately, BSD and the Single UNIX Specification disagree on the prototype of the function. The BSD variant receives an `unsigned long` and has no return value. The SUS variant, however, defines `usleep()` to accept a `useconds_t` type and return an `int`. Linux follows SUS if `_XOPEN_SOURCE` is defined as `500` or higher. If `_XOPEN_SOURCE` is undefined or set to less than `500`, Linux follows BSD.

The SUS version returns `0` on success and `-1` on error. Valid `errno` values are `EINTR`, if the nap was interrupted by a signal, or `EINVAL`, if `usecs` was too large (on Linux, the full range of the type is valid, and thus this error will never occur).

According to the specification, the `useconds_t` type is an unsigned integer capable of holding values as high as 1,000,000.

Due to the differences between the conflicting prototypes and the fact that some Unix systems may support one or the other, but not both, it is wise never to explicitly include the `useconds_t` type in your code. For maximum portability, assume that the parameter is an `unsigned int`, and do not rely on `usleep()`'s return value:

```
void usleep (unsigned int usec);
```

Usage is then:

```
unsigned int usecs = 200;

usleep (usecs);
```

This works with either variant of the function, and checking for errors is still possible:

```
errno = 0;
usleep (1000);
if (errno)
        perror ("usleep");
```

Most programs, however, do not check for or care about `usleep()` errors.

Sleeping with Nanosecond Resolution

Linux deprecates the `usleep()` function, replacing it with `nanosleep()`, which provides nanosecond resolution and a smarter interface:

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

int nanosleep (const struct timespec *req,
               struct timespec *rem);
```

A successful call to `nanosleep()` puts the invoking process to sleep for the time specified by `req` and then returns `0`. On error, the call returns `-1` and sets `errno` appropriately. If a signal interrupts the sleep, the call can return before the specified time has elapsed. In that case, `nanosleep()` returns `-1`, and sets `errno` to `EINTR`. If `rem` is not `NULL`, the function places the remaining time to sleep (the amount of `req` not slept) in `rem`. The program may then reissue the call, passing `rem` for `req` (as shown later in this section).

Here are the other possible `errno` values:

`EFAULT`
: `req` or `rem` is an invalid pointer.

`EINVAL`
: One of the fields in `req` is invalid.

In the basic case, usage is simple:

```
struct timespec req = { .tv_sec = 0,
                        .tv_nsec = 200 };

/* sleep for 200 ns */
ret = nanosleep (&req, NULL);
if (ret)
        perror ("nanosleep");
```

And here is an example using the second parameter to continue the sleep if interrupted:

```
struct timespec req = { .tv_sec = 0,
                        .tv_nsec = 1369 };
struct timespec rem;
int ret;

/* sleep for 1369 ns */
retry:
ret = nanosleep (&req, &rem);
if (ret) {
        if (errno == EINTR) {
                /* retry, with the provided time remaining */
```

```
                req.tv_sec = rem.tv_sec;
                req.tv_nsec = rem.tv_nsec;
                goto retry;
        }
        perror ("nanosleep");
}
```

Finally, here's an alternative approach (perhaps more efficient, but less readable) toward the same goal:

```
struct timespec req = { .tv_sec = 1,
                        .tv_nsec = 0 };
struct timespec rem, *a = &req, *b = &rem;

/* sleep for 1s */
while (nanosleep (a, b) && errno == EINTR) {
        struct timespec *tmp = a;
        a = b;
        b = tmp;
}
```

`nanosleep()` has several advantages over `sleep()` and `usleep()`:

- Nanosecond, as opposed to second or microsecond, resolution
- Standardized by POSIX.1b
- Not implemented via signals (the pitfalls of which are discussed later)

Despite deprecation, many programs prefer to use `usleep()` rather than `nanosleep()`. Because `+nanosleep()` is a POSIX standard and does not use signals, new programs should prefer it (or the interface discussed in the next section) to `sleep()` and `usleep()`.

An Advanced Approach to Sleep

As with all of the classes of time functions we have thus far studied, the POSIX clocks family provides the most advanced sleep interface:

```
#include <time.h>

int clock_nanosleep (clockid_t clock_id,
                     int flags,
                     const struct timespec *req,
                     struct timespec *rem);
```

`clock_nanosleep()` behaves similarly to `nanosleep()`. In fact, this call:

```
ret = nanosleep (&req, &rem);
```

is the same as this call:

```
ret = clock_nanosleep (CLOCK_REALTIME, 0, &req, &rem);
```

The difference lies in the `clock_id` and `flags` parameters. The former specifies the time source to measure against. Most time sources are valid, although you cannot specify the CPU clock of the invoking process (e.g., `CLOCK_PROCESS_CPUTIME_ID`); doing so would make no sense because the call suspends execution of the process, and thus the process time stops increasing.

What time source you specify depends on your program's goals for sleeping. If you are sleeping until some absolute time value, `CLOCK_REALTIME` may make the most sense. If you are sleeping for a relative amount of time, `CLOCK_MONOTONIC` definitely is the ideal time source.

The `flags` parameter is either `TIMER_ABSTIME` or `0`. If it is `TIMER_ABSTIME`, the value specified by `req` is treated as absolute and not relative. This solves a potential race condition. To explain the value of this parameter, assume that a process, at time `T+0`, wants to sleep until time `T+1`. At `T+0`, the process calls `clock_gettime()` to obtain the current time (`T+0`). It then subtracts `T+0` from `T+1`, obtaining `Y`, which it passes to `clock_nanosleep()`. Some amount of time, however, will have passed between the moment at which the time was obtained and the moment at which the process goes to sleep. Worse, what if the process was scheduled off the processor, incurred a page fault, or something similar? There is always a potential race condition in between obtaining the current time, calculating the time differential, and actually sleeping.

The `TIMER_ABSTIME` flag nullifies the race by allowing a process to directly specify `T+1`. The kernel suspends the process until the specified time source reaches `T1 + 1`. If the specified time source's current time already exceeds `T+1`, the call returns immediately.

Let's look at both relative and absolute sleeping. The following example sleeps for 1.5 seconds:

```
struct timespec ts = { .tv_sec = 1, .tv_nsec = 500000000 };
int ret;

ret = clock_nanosleep (CLOCK_MONOTONIC, 0, &ts, NULL);
if (ret)
        perror ("clock_nanosleep");
```

Conversely, the next example sleeps until an absolute value of time—which is exactly one second from what the `clock_gettime()` call returns for the `CLOCK_MONOTONIC` time source—is reached:

```
struct timespec ts;
int ret;

/* we want to sleep until one second from NOW */
ret = clock_gettime (CLOCK_MONOTONIC, &ts);
if (ret) {
        perror ("clock_gettime");
        return;
```

```
}

ts.tv_sec += 1;
printf ("We want to sleep until sec=%ld nsec=%ld\n",
        ts.tv_sec, ts.tv_nsec);
ret = clock_nanosleep (CLOCK_MONOTONIC, TIMER_ABSTIME,
                       &ts, NULL);
if (ret)
        perror ("clock_nanosleep");
```

Most programs need only a relative sleep because their sleep needs are not very strict. Some real-time processes, however, have very exact timing requirements and need the absolute sleep to avoid the danger of a potentially devastating race condition.

A Portable Way to Sleep

Recall from Chapter 2 our friend `select()`:

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

int select (int n,
            fd_set *readfds,
            fd_set *writefds,
            fd_set *exceptfds,
            struct timeval *timeout);
```

As mentioned in that chapter, `select()` provides a portable way to sleep with subsecond resolution. For a long time, portable Unix programs were stuck with `sleep()` for their naptime needs: `usleep()` was not widely available, and `nanosleep()` was as of yet unwritten. Developers discovered that passing `select()` `0` for `n`, `NULL` for all three of the `fd_set` pointers, and the desired sleep duration for `timeout` resulted in a portable and efficient way to put processes to sleep:

```
struct timeval tv = { .tv_sec = 0,
                      .tv_usec = 757 };

/* sleep for 757 us */
select (0, NULL, NULL, NULL, &tv);
```

If portability to older Unix systems is a concern, using `select()` may be your best bet.

Overruns

All of the interfaces discussed in this section guarantee that they will sleep *at least as long as requested* (or return an error indicating otherwise). They will never return success without the requested delay elapsing. It is possible, however, for an interval *longer* than the requested delay to pass.

This phenomenon may be due to simple scheduling behavior—the requested time may have elapsed, and the kernel may have woken up the process on time, but the scheduler may have selected a different task to run.

There exists a more insidious cause, however: *timer overruns*. This occurs when the granularity of the timer is coarser than the requested time interval. For example, assume the system timer ticks in 10 ms intervals and a process requests a 1 ms sleep. The system is able to measure time and respond to time-related events (such as waking up a process from sleep) only at 10 ms intervals. If, when the process issues the sleep request, the timer is 1 ms away from a tick, everything will be fine—in 1 ms, the requested time (1 ms) will elapse, and the kernel will wake up the process. If, however, the timer hits right as the process requests the sleep, there won't be another timer tick for 10 ms. Subsequently, the process will sleep an extra 9 ms! That is, there will be nine 1 ms overruns. On average, a timer with a period of *X* has an overrun rate of *X/2*.

The use of high-precision time sources, such as those provided by POSIX clocks, and higher values for `HZ`, minimize timer overrun.

Alternatives to Sleeping

If possible, you should avoid sleeping. Often, you cannot, and that's fine—particularly if your code is sleeping for less than a second. Code that is laced with sleeps in order to "busy-wait" for events is usually of poor design. Code that blocks on a file descriptor, allowing the kernel to handle the sleep and wake up the process, is better. Instead of the process spinning in a loop until the event hits, the kernel can block the process from execution and wake it up only when needed.

Timers

Timers provide a mechanism for notifying a process when a given amount of time elapses. The amount of time before a timer *expires* is called the *delay*, or the *expiration*. How the kernel notifies the process that the timer has expired depends on the timer. The Linux kernel offers several types. We will study them all.

Timers are useful for several reasons. Examples include refreshing the screen 60 times per second or canceling a pending transaction if it is still ongoing after 500 milliseconds.

Simple Alarms

`alarm()` is the simplest timer interface:

```
#include <unistd.h>

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

A call to this function schedules the delivery of a `SIGALRM` signal to the invoking process after `seconds` of real time have elapsed. If a previously scheduled signal was pending, the call cancels the alarm, replaces it with the newly requested alarm, and returns the number of seconds remaining in the previous alarm. If `seconds` is `0`, the previous alarm, if any, is canceled, but no new alarm is scheduled.

Successful use of this function thus also requires registering a signal handler for the `SIGALRM` signal. (Signals and signal handlers were covered in the previous chapter.) Here is a code snippet that registers a `SIGALRM` handler, `alarm_handler()`, and sets an alarm for five seconds:

```
void alarm_handler (int signum)
{
        printf ("Five seconds passed!\n");
}

void func (void)
{
        signal (SIGALRM, alarm_handler);
        alarm (5);

        pause ();
}
```

Interval Timers

Interval timer system calls, which first appeared in 4.2BSD, have since been standardized in POSIX and provide more control than `alarm()`:

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

int getitimer (int which,
               struct itimerval *value);

int setitimer (int which,
               const struct itimerval *value,
               struct itimerval *ovalue);
```

Interval timers operate like `alarm()`, but optionally can automatically rearm themselves and operate in one of three distinct modes:

`ITIMER_REAL`
: Measures real time. When the specified amount of real time has elapsed, the kernel sends the process a `SIGALRM` signal.

`ITIMER_VIRTUAL`
: Decrements only while the process's user-space code is executing. When the specified amount of process time has elapsed, the kernel sends the process a `SIGVTALRM`.

`ITIMER_PROF`
: Decrements both while the process is executing, and while the kernel is executing on behalf of the process (for example, completing a system call). When the specified amount of time has elapsed, the kernel sends the process a `SIGPROF` signal. This mode is usually coupled with `ITIMER_VIRTUAL` so that the program can measure user and kernel time spent by the process.

`ITIMER_REAL`
: Measures the same time as `alarm()`; the other two modes are useful for profiling.

The `itimerval` structure allows the user to specify the amount of time until the timer expires, as well as the expiration, if any, with which to rearm the timer upon expiration:

```
struct itimerval {
        struct timeval it_interval;  /* next value */
        struct timeval it_value;     /* current value */
};
```

Recall from earlier that the `timeval` structure provides microsecond resolution:

```
struct timeval {
        long tv_sec;   /* seconds */
        long tv_usec;  /* microseconds */
};
```

`setitimer()` arms a timer of type `which` with the expiration specified by `it_value`. Once the time specified by `it_value` elapses, the kernel rearms the timer with the time provided by `it_interval`. Thus, `it_value` is the time remaining on the current timer. Once `it_value` reaches zero, it is set to `it_interval`. If the timer expires, and `it_interval` is `0`, the timer is not rearmed. Similarly, if an active timer's `it_value` is set to `0`, the timer is stopped and not rearmed.

If `ovalue` is not `NULL`, the previous values for the interval timer of type `which` are returned.

`getitimer()` returns the current values for the interval timer of type `which`.

Both functions return `0` on success and `-1` on error, in which case `errno` is set to one of the following:

`EFAULT`
: `value` or `ovalue` is an invalid pointer.

`EINVAL`
: `which` is not a valid interval timer type.

The following code snippet creates a `SIGALRM` signal handler (again, see Chapter 10) and then arms an interval timer with an initial expiration of five seconds, followed by a subsequent interval of one second:

```
void alarm_handler (int signo)
{
        printf ("Timer hit!\n");
}

void foo (void)
{
        struct itimerval delay;
        int ret;

        signal (SIGALRM, alarm_handler);

        delay.it_value.tv_sec = 5;
        delay.it_value.tv_usec = 0;
        delay.it_interval.tv_sec = 1;
        delay.it_interval.tv_usec = 0;
        ret = setitimer (ITIMER_REAL, &delay, NULL);
        if (ret) {
                perror ("setitimer");
                return;
        }

        pause ();
}
```

Some Unix systems implement `sleep()` and `usleep()` via `SIGALRM`. `alarm()` and `setitimer()` also use `SIGALRM`. Therefore, programmers must be careful not to overlap calls to these functions; the results are undefined. For the purpose of brief waits, programmers should use `nanosleep()`, which POSIX dictates will not use signals. For timers, programmers should use `setitimer()` *or* `alarm()`.

Advanced Timers

The most powerful timer interface, not surprisingly, hails from the POSIX clocks family.

With POSIX clocks–based timers, the acts of instantiating, initializing, and ultimately deleting a timer are separated into three different functions: `timer_create()` creates the timer, `timer_settime()` initializes the timer, and `timer_delete()` destroys it.

The POSIX clocks family of timer interfaces is undoubtedly the most advanced but also the newest (ergo the least portable) and most complicated to use. If simplicity or portability is a prime motivator, `setitimer()` is most likely a better choice.

Creating a timer

To create a timer, use `timer_create()`:

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

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

A successful call to `timer_create()` creates a new timer associated with the POSIX clock `clockid`, stores a unique timer identification in `timerid`, and returns `0`. This call merely sets up the conditions for running the timer; nothing actually happens until the timer is armed, as shown in the following section.

The following example creates a new timer keyed off the `CLOCK_PROCESS_CPUTIME_ID` POSIX clock and stores the timer's ID in `timer`:

```
timer_t timer;
int ret;

ret = timer_create (CLOCK_PROCESS_CPUTIME_ID,
                    NULL,
                    &timer);
if (ret)
        perror ("timer_create");
```

On failure, the call returns `-1`, `timerid` is undefined, and the call sets `errno` to one of the following:

`EAGAIN`
: The system lacks sufficient resources to complete the request.

`EINVAL`
: The POSIX clock specified by `clockid` is invalid.

`ENOTSUP`
: The POSIX clock specified by `clockid` is valid, but the system does not support using the clock for timers. POSIX guarantees that all implementations support the `CLOCK_REALTIME` clock for timers. Whether other clocks are supported is up to the implementation.

The `evp` parameter, if non-`NULL`, defines the asynchronous notification that occurs when the timer expires. The header `<signal.h>` defines the structure. Its contents are supposed to be opaque to the programmer, but it has at least the following fields:

```
#include <signal.h>

struct sigevent {
        union sigval sigev_value;
        int sigev_signo;
        int sigev_notify;
        void (*sigev_notify_function)(union sigval);
        pthread_attr_t *sigev_notify_attributes;
```

```
};

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

POSIX clocks–based timers allow much greater control over how the kernel notifies the process when a timer expires, allowing the process to specify exactly which signal the kernel will emit, or even allowing the kernel to spawn a thread and execute a function in response to timer expiration. A process specifies the behavior on timer expiration via `sigev_notify`, which must be one of the following three values:

`SIGEV_NONE`
: A "null" notification. On timer expiration, nothing happens.

`SIGEV_SIGNAL`
: On timer expiration, the kernel sends the process the signal specified by `sigev_signo`. In the signal handler, `si_value` is set to `sigev_value`.

`SIGEV_THREAD`
: On timer expiration, the kernel spawns a new thread (within this process) and has it execute `sigev_notify_function`, passing `sigev_value` as its sole argument. The thread terminates when it returns from this function. If `sigev_notify_attributes` is not `NULL`, the provided `pthread_attr_t` structure defines the behavior of the new thread.

If `evp` is `NULL`, as it was in our earlier example, the timer's expiration notification is set up as if `sigev_notify` were `SIGEV_SIGNAL`, `sigev_signo` were `SIGALRM`, and `sigev_value` were the timer's ID. Thus, by default, these timers notify in a manner similar to POSIX interval timers. Via customization, though, they can do much, much more!

The following example creates a timer keyed off `CLOCK_REALTIME`. When the timer expires, the kernel will issue the `SIGUSR1` signal and set `si_value` to the address storing the timer's ID:

```
struct sigevent evp;
timer_t timer;
int ret;

evp.sigev_value.sival_ptr = &timer;
evp.sigev_notify = SIGEV_SIGNAL;
evp.sigev_signo = SIGUSR1;
ret = timer_create (CLOCK_REALTIME,
                    &evp,
                    &timer);
if (ret)
        perror ("timer_create");
```

Arming a timer

A timer created by `timer_create()` is unarmed. To associate it with an expiration and start the clock ticking, use `timer_settime()`:

```
#include <time.h>

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

A successful call to `timer_settime()` arms the timer specified by `timerid` with the expiration `value`, which is an `itimerspec` structure:

```
struct itimerspec {
        struct timespec it_interval;  /* next value */
        struct timespec it_value;     /* current value */
};
```

As with `setitimer()`, `it_value` specifies the current timer expiration. When the timer expires, `it_value` is refreshed with the value from `it_interval`. If `it_interval` is `0`, the timer is not an interval timer and will disarm once `it_value` expires.

Recall from earlier that the `timespec` structure provides nanosecond resolution:

```
struct timespec {
        time_t  tv_sec;       /* seconds */
        long    tv_nsec;      /* nanoseconds */
};
```

If `flags` is `TIMER_ABSTIME`, the time specified by `value` is interpreted as absolute (as opposed to the default interpretation, where the value is relative to the current time). This modified behavior prevents a race condition during the steps of obtaining the current time, calculating the relative difference between that time, and a desired future time, and arming the timer. See the discussion in the earlier section, "An Advanced Approach to Sleep" on page 383 for details.

If `ovalue` is non-`NULL`, the previous timer expiration is saved in the provided `itimerspec`. If the timer was previously disarmed, the structure's members are all set to `0`.

Using the `timer` value initialized earlier by `timer_create()`, the following example creates a periodic timer that expires every second:

```
struct itimerspec ts;
int ret;

ts.it_interval.tv_sec = 1;
ts.it_interval.tv_nsec = 0;
ts.it_value.tv_sec = 1;
ts.it_value.tv_nsec = 0;
```

```
ret = timer_settime (timer, 0, &ts, NULL);
if (ret)
        perror ("timer_settime");
```

Obtaining the expiration of a timer

You can get the expiration time of a timer without resetting it via `timer_gettime()`:

```
#include <time.h>

int timer_gettime (timer_t timerid,
                   struct itimerspec *value);
```

A successful call to `timer_gettime()` stores the expiration time of the timer specified by `timerid` in the structure pointed at by `value` and returns `0`. On failure, the call returns `-1` and sets `errno` to one of the following:

`EFAULT`
: `value` is an invalid pointer.

`EINVAL`
: `timerid` is an invalid timer.

For example:

```
struct itimerspec ts;
int ret;

ret = timer_gettime (timer, &ts);
if (ret)
        perror ("timer_gettime");
else {
        printf ("current sec=%ld nsec=%ld\n",
                ts.it_value.tv_sec, ts.it_value.tv_nsec);
        printf ("next sec=%ld nsec=%ld\n",
                ts.it_interval.tv_sec, ts.it_interval.tv_nsec);
}
```

Obtaining the overrun of a timer

POSIX defines an interface for determining how many, if any, overruns occurred on a given timer:

```
#include <time.h>

int timer_getoverrun (timer_t timerid);
```

On success, `timer_getoverrun()` returns the number of additional timer expirations that have occurred between the initial expiration of the timer and notification to the process—for example, via a signal—that the timer expired. For instance, in our earlier example, where a 1 ms timer ran for 10 ms, the call would return `9`.

According to POSIX, if the number of overruns is equal to or greater than `DELAYTIMER_MAX`, the call returns `DELAYTIMER_MAX`. Unfortunately, Linux does not implement this behavior: instead, once the number of timer overruns exceeds `DELAYTIMER_MAX`, it wraps back to zero and starts anew.

On failure, the function returns `-1` and sets `errno` to `EINVAL`, the lone error condition, signifying that the timer specified by `timerid` is invalid.

For example:

```
int ret;

ret = timer_getoverrun (timer);
if (ret == -1)
        perror ("timer_getoverrun");
else if (ret == 0)
        printf ("no overrun\n");
else
        printf ("%d overrun(s)\n", ret);
```

Deleting a timer

Deleting a timer is easy:

```
#include <time.h>

int timer_delete (timer_t timerid);
```

A successful call to `timer_delete()` destroys the timer associated with `timerid` and returns `0`. On failure, the call returns `-1`, and `errno` is set to `EINVAL`, the lone error condition, signifying that `timerid` is not a valid timer.

APPENDIX A

GCC Extensions to the C Language

The GNU Compiler Collection (GCC) provides many extensions to the C language, some of which have proven to be of particular value to system programmers. The majority of the additions to the C language that we'll cover in this appendix offer ways for programmers to provide additional information to the compiler about the behavior and intended use of their code. The compiler, in turn, utilizes this information to generate more efficient machine code. Other extensions fill in gaps in the C programming language, particularly at lower levels.

GCC provides several extensions now available in the latest C standard, ISO C11. Some of these extensions function similarly to their C11 cousins, but ISO C11 implemented other extensions rather differently. New code should use the standardized variants of these features. We won't cover such extensions here; we'll discuss only GCC-unique additions.

GNU C

The flavor of C supported by GCC is often called GNU C. In the 1990s, GNU C filled in several gaps in the C language, providing features such as complex variables, zero-length arrays, inline functions, and named initializers. But after nearly a decade, C was finally upgraded, and with the standardization of ISO C99 and then ISO C11, GNU C extensions grew less relevant. Nonetheless, GNU C continues to provide useful features, and many Linux programmers still use a subset of GNU C—often just an extension or two—in their C99- or C11-compliant code.

One prominent example of a GCC-specific code base is the Linux kernel, which is written strictly in GNU C. Recently, however, Intel has invested engineering effort in allowing the Intel C Compiler (ICC) to understand the GNU C extensions used by the kernel. Consequently, many of these extensions are now growing less GCC-specific.

Inline Functions

The compiler copies the entire code of an "inline" function into the site where the function is called. Instead of storing the function externally and jumping to it whenever it is called, it runs the contents of the function directly. Such behavior saves the overhead of the function call and allows for potential optimizations at the call site because the compiler can optimize the caller and callee together. This latter point is particularly valid if the parameters to the function are constant at the call site. Naturally, however, copying a function into each and every chunk of code that invokes it can have a detrimental effect on code size. Therefore, functions should be inlined only if they are small and simple or are not called in many different places.

For many years, GCC has supported the `inline` keyword, instructing the compiler to inline the given function. C99 formalized this keyword:

```
static inline int foo (void) { /* ... */ }
```

Technically, however, the keyword is merely a hint—a suggestion to the compiler to consider inlining the given function. GCC further provides an extension for instructing the compiler to *always* inline the designated function:

```
static inline __attribute__ ((always_inline)) int foo (void) { /* ... */ }
```

The most obvious candidate for an inline function is a preprocessor macro. An inline function in GCC will perform as well as a macro and receives type checking. For example, instead of this macro:

```
#define max(a,b) ({ a > b ? a : b; })
```

one might use the corresponding inline function:

```
static inline max (int a, int b)
{
        if (a > b)
                return a;
        return b;
}
```

Programmers tend to overuse inline functions. Function call overhead on most modern architectures—x86 in particular—is very, very low. Only the most worthy of functions should receive consideration!

Suppressing Inlining

In its most aggressive optimization mode, GCC automatically selects functions that appear suitable for inlining and inlines them. This is normally a good idea, but sometimes the programmer knows that a function will perform incorrectly if inlined. One example of this is when using `__builtin_return_address` (discussed later in this appendix). To suppress inlining, use the `noinline` keyword:

```
__attribute__ ((noinline)) int foo (void) { /* ... */ }
```

Pure Functions

A "pure" function is one that has no side effects and whose return value reflects only the function's parameters or nonvolatile global variables. Any parameter or global variable access must be read-only. Loop optimization and subexpression elimination can be applied to such functions. Functions are marked as pure via the `pure` keyword:

```
__attribute__ ((pure)) int foo (int val) { /* ... */ }
```

A common example is `strlen()`. Given identical inputs, this function's return value is invariant across multiple invocations, and thus it can be pulled out of a loop and called just once. For example, consider the following code:

```
/* character by character, print each letter in 'p' in uppercase */
for (i = 0; i < strlen (p); i++)
        printf ("%c", toupper (p[i]));
```

If the compiler does not know that `strlen()` is pure, it would need to invoke the function with each iteration of the loop.

Smart programmers—as well as the compiler, if `strlen()` were marked pure—would write or generate code like this:

```
size_t len;

len = strlen (p);
for (i = 0; i < len; i++)
        printf ("%c", toupper (p[i]));
```

Parenthetically, even smarter programmers (such as this book's readers) would write:

```
while (*p)
        printf ("%c", toupper (*p++));
```

It is illegal and indeed makes no sense for a pure function to return `void`, as the return value is the sole point of such functions. An example of a nonpure function is `random()`.

Constant Functions

A "constant" function is a stricter variant of a pure function. Such functions cannot access global variables and cannot take pointers as parameters. Thus, the constant function's return value reflects nothing but the passed-by-value parameters. Additional optimizations, on top of those possible with pure functions, are possible for such functions. Math functions, such as `abs()`, are examples of constant functions (presuming they don't save state or otherwise pull tricks in the name of optimization). A programmer marks a function constant via the `const` keyword:

```
__attribute__ ((const)) int foo (int val) { /* ... */ }
```

As with pure functions, it makes no sense for a constant function to return `void`.

Functions That Do Not Return

If a function does not return, perhaps because it invariantly calls `exit()`, the programmer can mark the function with the `noreturn` keyword, enlightening the compiler to that fact:

```
__attribute__ ((noreturn)) void foo (int val) { /* ... */ }
```

In turn, the compiler can make additional optimizations, with the understanding that under no circumstances will the invoked function ever return. It does not make sense for such a function to return anything but `void`.

Functions That Allocate Memory

If a function returns a pointer that can never alias[1] existing memory—almost assuredly because the function just allocated fresh memory, and is returning a pointer to it—the programmer can mark the function as such with the `malloc` keyword, and the compiler can in turn perform suitable optimizations:

```
__attribute__ ((malloc)) void * get_page (void)
{
        int page_size;

        page_size = getpagesize ();
        if (page_size <= 0)
                return NULL;

        return malloc (page_size);
}
```

Forcing Callers to Check the Return Value

Not an optimization, but a programming aid, the `warn_unused_result` attribute instructs the compiler to generate a warning whenever the return value of a function is not stored or used in a conditional statement:

```
__attribute__ ((warn_unused_result)) int foo (void) { /* ... */ }
```

1. A memory *alias* occurs when two or more pointer variables point at the same memory address. This can happen in trivial cases where a pointer is assigned the value of another pointer and also in more complex, less obvious cases. If a function is returning the address of newly allocated memory, no other pointers to that same address should exist.

This allows the programmer to ensure that all callers check and handle the return value from a function where the value is of particular importance. Functions with important but oft-ignored return values, such as `read()`, make excellent candidates for this attribute. Such functions cannot return `void`.

Marking Functions as Deprecated

The `deprecated` attribute instructs the compiler to generate a warning at the call site whenever the function is invoked:

```
__attribute__ ((deprecated)) void foo (void) { /* ... */ }
```

This helps wean programmers off deprecated and obsolete interfaces.

Marking Functions as Used

Occasionally, no code visible to a compiler invokes a particular function. Marking a function with the `used` attribute instructs the compiler that the program uses that function, despite appearances that the function is never referenced:

```
static __attribute__ ((used)) void foo (void) { /* ... */ }
```

The compiler therefore outputs the resulting assembly language and does not display a warning about an unused function. This attribute is useful if a static function is invoked only from handwritten assembly code. Normally, if the compiler is not aware of any invocation, it will generate a warning, and potentially optimize away the function.

Marking Functions or Parameters as Unused

The `unused` attribute tells the compiler that the given function or function parameter is unused and instructs it not to issue any corresponding warnings:

```
int foo (long __attribute__ ((unused)) value) { /* ... */ }
```

This is useful if you're compiling with `-W` or `-Wunused` and you want to catch unused function parameters, but you occasionally have functions that must match a predetermined signature (as is common in event-driven GUI programming or signal handlers).

Packing a Structure

The `packed` attribute tells the compiler that a type or variable should be packed into memory using the minimum amount of space possible, potentially disregarding alignment requirements. If specified on a `struct` or `union`, all variables therein are so packed. If specified on just one variable, only that specific object is packed.

The following packs all variables within the structure into the minimum amount of space:

```
struct __attribute__ ((packed)) foo { ... };
```

As an example, a structure containing a `char` followed by an `int` would most likely find the integer aligned to a memory address not immediately following the `char`, but, say, three bytes later. The compiler aligns the variables by inserting bytes of unused padding between them. A packed structure lacks this padding, potentially consuming less memory but failing to meet architectural alignment requirements.

Increasing the Alignment of a Variable

As well as allowing packing of variables, GCC also allows programmers to specify an alternative minimum alignment for a given variable. GCC will then align the specified variable to *at least* this value, as opposed to the minimum required alignment dictated by the architecture and ABI. For example, this statement declares an integer named `beard_length` with a minimum alignment of 32 bytes (as opposed to the typical alignment of 4 bytes on machines with 32-bit integers):

```
int beard_length __attribute__ ((aligned (32))) = 0;
```

Forcing the alignment of a type is generally useful only when dealing with hardware that may impose greater alignment requirements than the architecture itself, or when you are hand-mixing C and assembly code and you want to use instructions that require specially aligned values. One example where this alignment functionality is utilized is for storing oft-used variables on processor cache lines to optimize cache behavior. The Linux kernel makes use of this technique.

As an alternative to specifying a certain minimum alignment, you can ask that GCC align a given type to the largest minimum alignment that is ever used for any data type. For example, this instructs GCC to align `parrot_height` to the largest alignment it ever uses, which is probably the alignment of a `double`:

```
short parrot_height __attribute__ ((aligned)) = 5;
```

This decision generally involves a space/time trade-off: variables aligned in this manner consume more space, but copying to or from them (along with other complex manipulations) may be faster because the compiler can issue machine instructions that deal with the largest amount of memory.

Various aspects of the architecture or the system's toolchain may impose maximum limits on a variable's alignment. For example, on some Linux architectures, the linker is unable to recognize alignments beyond a rather small default. In that case, an alignment provided using this keyword is rounded down to the smallest allowed alignment. For example, if you request an alignment of 32, but the system's linker is unable to align to more than 8 bytes, the variable will be aligned along an 8-byte boundary.

Placing Global Variables in a Register

GCC allows programmers to place global variables in a specific machine register, where the variables will then reside for the duration of the program's execution. GCC calls such variables *global register variables*.

The syntax requires that the programmer specify the machine register. The following example uses `ebx`:

```
register int *foo asm ("ebx");
```

The programmer must select a variable that is not function-clobbered: that is, the selected variable must be usable by local functions, saved and restored on function call invocation, and not specified for any special purpose by the architecture or operating system's ABI. The compiler will generate a warning if the selected register is inappropriate. If the register is appropriate—`ebx`, used in this example, is fine for the x86 architecture—the compiler will in turn stop using the register itself.

Such an optimization can provide huge performance boosts if the variable is frequently used. A good example is with a virtual machine. Placing the variable that holds, say, the virtual stack frame pointer in a register might lead to substantial gains. On the other hand, if the architecture is starved of registers to begin with (as the x86 architecture is), this optimization makes little sense.

Global register variables cannot be used in signal handlers or by more than one thread of execution. They also cannot have initial values because there is no mechanism for executable files to supply default contents for registers. Global register variable declarations should precede any function definitions.

Branch Annotation

GCC allows programmers to annotate the expected value of an expression—for example, to tell the compiler whether a conditional statement is likely to be true or false. GCC, in turn, can then perform block reordering and other optimizations to improve the performance of conditional branches.

The GCC syntax for branch notation is horrendously ugly. To make branch annotation easier on the eyes, we use preprocessor macros:

```
#define likely(x)    __builtin_expect (!!(x), 1)
#define unlikely(x)  __builtin_expect (!!(x), 0)
```

Programmers can mark an expression as likely or unlikely true by wrapping it in `likely()` or `unlikely()`, respectively.

The following example marks a branch as unlikely true (that is, likely to be false):

```
int ret;

ret = close (fd);
if (unlikely (ret))
        perror ("close");
```

Conversely, the following example marks a branch as likely true:

```
const char *home;

home = getenv ("HOME");
if (likely (home))
        printf ("Your home directory is %s\n", home);
else
        fprintf (stderr, "Environment variable HOME not set!\n");
```

As with inline functions, programmers have a tendency to overuse branch annotation. Once you start anointing expressions, you might be tempted to mark *all* expressions. Be careful, though—you should mark branches as likely or unlikely only if you know *a priori* and with little doubt that the expressions will be true or false *nearly all of the time* (say, with 99 percent certainty). Seldom-occurring errors are good candidates for `unlikely()`. Keep in mind that a false prediction is worse than no prediction at all.

Getting the Type of an Expression

GCC provides the `typeof()` keyword to obtain the type of a given expression. Semantically, the keyword operates the same as `sizeof()`. For example, this expression returns the type of whatever `x` points at:

```
typeof (*x)
```

We can use this to declare an array, `y`, of those types:

```
typeof (*x) y[42];
```

A popular use for `typeof()` is to write "safe" macros that can operate on any arithmetic value and evaluate their parameters only once:

```
#define max(a,b) ({          \
        typeof (a) _a = (a); \
        typeof (b) _b = (b); \
       _a > _b ? _a : _b; \
})
```

Getting the Alignment of a Type

GCC provides the keyword *`alignof`* to obtain the alignment of a given object. The value is architecture- and ABI-specific. If the current architecture does not have a required alignment, the keyword returns the ABI's recommended alignment. Otherwise, the keyword returns the minimum required alignment.

The syntax is identical to `sizeof()`:

```
__alignof__(int)
```

Depending on the architecture, this probably returns 4, as 32-bit integers are generally aligned along 4-byte boundaries.

alignof() in C11 and C++11

C11 and C++11 introduced `alignof()`, which works identically to `alignof()` but is standardized. If writing a C11 or C++11 program, prefer `alignof()`.

The keyword works on lvalues, too. In that case, the returned alignment is the minimum alignment of the backing type, not the actual alignment of the specific lvalue. If the minimum alignment was changed via the `aligned` attribute (described earlier, in "Increasing the Alignment of a Variable" on page 400), that change is reflected by `__alignof__`.

For example, consider this structure:

```
struct ship {
        int year_built;
        char cannons;
        int mast_height;
};
```

along with this code snippet:

```
struct ship my_ship;

printf ("%d\n", __alignof__(my_ship.cannons));
```

The *`alignof`* in this snippet will return 1, even though structure padding probably results in `cannons` consuming 4 bytes.

The Offset of a Member Within a Structure

GCC provides a built-in keyword for obtaining the offset of a member of a structure within that structure. The `offsetof()` macro, defined in `<stddef.h>`, is part of the ISO C standard. Most definitions are horrid, involving obscene pointer arithmetic and code unfit for minors. The GCC extension is simpler and potentially faster:

```
#define offsetof(type, member)  __builtin_offsetof (type, member)
```

A call returns the offset of `member` within `type`—that is, the number of bytes, starting from zero, from the beginning of the structure to that member. For example, consider the following structure:

```
struct rowboat {
        char *boat_name;
        unsigned int nr_oars;
        short length;
};
```

The actual offsets depend on the size of the variables and the architecture's alignment requirements and padding behavior. On a 32-bit machine, we might expect calling `offsetof()` on `struct rowboat` with `boat_name`, `nr_oars`, and `length` to return `0`, `4`, and `8`, respectively.

On a Linux system, the `offsetof()` macro should be defined using the GCC keyword and need not be redefined.

Obtaining the Return Address of a Function

GCC provides a keyword for obtaining the return address of the current function, or one of the callers of the current function:

```
void * __builtin_return_address (unsigned int level)
```

The parameter `level` specifies the function in the call chain whose address should be returned. A value of `0` asks for the return address of the current function, a value of `1` asks for the return address of the caller of the current function, a value of `2` asks for *that* function's caller's return address, and so on.

If the current function is an inline function, the address returned is that of the calling function. If this is unacceptable, use the `noinline` keyword (described earlier, in "Suppressing Inlining" on page 396) to force the compiler not to inline the function.

There are several uses for the `__builtin_return_address` keyword. One is for debugging or informational purposes. Another is to unwind a call chain in order to implement introspection, a crash dump utility, a debugger, and so on.

Note that some architectures can return only the address of the invoking function. On such architectures, a nonzero parameter value can result in a random return value. Thus, any parameter other than `0` is nonportable and should be used only for debugging purposes.

Case Ranges

GCC allows `case` statement labels to specify a range of values for a single block. The general syntax is as follows:

```
case low ... high:
```

For example:

```
switch (val) {
case 1 ... 10:
        /* ... */
        break;
case 11 ... 20:
        /* ... */
        break;
default:
        /* ... */
}
```

This functionality is quite useful for ASCII case ranges, too:

```
case 'A' ... 'Z':
```

Note that there should be a space before and after the ellipsis. Otherwise, the compiler can become confused, particularly with integer ranges. Always do the following:

```
case 4 ... 8:
```

and never this:

```
case 4...8:
```

Void and Function Pointer Arithmetic

In GCC, addition and subtraction operations are allowed on pointers of type `void` and pointers to functions. Normally, ISO C does not allow arithmetic on such pointers because the size of a "void" is a nonsensical concept and is dependent on what the pointer is actually pointing to. To facilitate such arithmetic, GCC treats the size of the referential object as one byte. Thus, the following snippet advances `a` by one:

```
a++;        /* a is a void pointer */
```

The option `-Wpointer-arith` causes GCC to generate a warning when these extensions are used.

More Portable and More Beautiful in One Fell Swoop

Let's face it, the *`attribute`* syntax is not pretty. Some of the extensions we've looked at in this chapter require preprocessor macros to make their use palatable, but all of them can benefit from a sprucing up in appearance.

With a little preprocessor magic, this is not hard. Further, in the same action, we can make the GCC extensions portable by defining them away in the case of a non-GCC compiler (whatever that is).

To do so, stick the following code snippet in a header and include that header in your source files:

```
#if __GNUC__  _ >= 3
# undef  inline
# define inline         inline __attribute__  _ ((always_inline))
# define __noinline     __attribute__ ((noinline))
# define __pure         __attribute__ ((pure))
# define __const        __attribute__ ((const))
# define __noreturn     __attribute__ ((noreturn))
# define __malloc       __attribute__ ((malloc))
# define __must_check   __attribute__ ((warn_unused_result))
# define __deprecated   __attribute__ ((deprecated))
# define __used         __attribute__ ((used))
# define __unused       __attribute__ ((unused))
# define __packed       __attribute__ ((packed))
# define __align(x)     __attribute__ ((aligned (x)))
# define __align_max    __attribute__ ((aligned))
# define likely(x)      __builtin_expect (!!(x), 1)
# define unlikely(x)    __builtin_expect (!!(x), 0)
#else
# define __noinline     /* no noinline */
# define __pure         /* no pure */
# define __const        /* no const */
# define __noreturn     /* no noreturn */
# define __malloc       /* no malloc */
# define __must_check   /* no warn_unused_result */
# define __deprecated   /* no deprecated */
# define __used         /* no used */
# define __unused       /* no unused */
# define __packed       /* no packed */
# define __align(x)     /* no aligned */
# define __align_max    /* no align_max */
# define likely(x)      (x)
# define unlikely(x)    (x)
#endif
```

For example, the following marks a function as pure, using our shortcut:

```
__pure int foo (void) { /* ... */ }
```

If GCC is in use, the function is marked with the `pure` attribute. If GCC is not the compiler, the preprocessor replaces the `__pure` token with a no-op. Note that you can place multiple attributes on a given definition, and thus you can use more than one of these defines on a single definition with no problems.

Easier, prettier, and portable!

APPENDIX B
Bibliography

This bibliography presents recommended reading related to system programming, broken down into four subcategories. None of these works are required reading. Instead, they represent my take on the top books on the given subject matter. If you find yourself pining for more information on the topics discussed here, these are my favorites.

Some of these books address material with which this book assumes the reader is already conversant, such as the C programming language. Other texts included make great supplements to this book, such as the works covering *gdb*, Git, or operating system design. Whatever the case, I recommend them all. Of course, these lists are certainly not exhaustive—please do explore other resources.

Books on the C Programming Language

These books document the C programming language, the lingua franca of system programming. If you do not code C as well as you speak your native tongue, one or more of the following works (coupled with a lot of practice!) ought to help you in that direction. If nothing else, the first title—universally known as *K&R*—is a treat to read. Its brevity reveals the simplicity of C.

The C Programming Language, 2nd ed. Brian Kernighan and Dennis Ritchie. Prentice Hall, 1988. This book, written by the author of the C programming language and his then coworker, is the bible of C programming.

C in a Nutshell. Peter Prinz and Tony Crawford. O'Reilly Media, 2005. A great book covering both the C language and the standard C library.

C Pocket Reference. Peter Prinz and Ulla Kirch-Prinz. Translated by Tony Crawford. O'Reilly Media, 2002. A concise reference to the C language, handily updated for ANSI C99.

Expert C Programming. Peter van der Linden. Prentice Hall, 1994. A wonderful discussion of lesser-known aspects of the C programming language, elucidated with an amazing wit and sense of humor. This book is rife with non sequiturs and jokes.

C Programming FAQs: Frequently Asked Questions, 2nd ed. Steve Summit. Addison-Wesley, 1995. This beast of a book contains more than 400 frequently asked questions (with answers) on the C programming language. Many of the FAQs beg obvious answers in the eyes of C masters, but some of the weightier questions and answers should impress even the most erudite of C programmers. Note there is an online version that has likely been more recently updated.

Books on Linux Programming

The following texts cover Linux programming, including discussions of topics not covered in this book and Linux programming tools.

Unix Network Programming, Volume 1: The Sockets Networking API. W. Richard Stevens et al. Addison-Wesley, 2003. The definitive tome on the socket API; unfortunately not specific to Linux, but fortunately recently updated for IPv6.

UNIX Network Programming, Volume 2: Interprocess Communications. W. Richard Stevens. Prentice Hall, 1998. An excellent discussion of interprocess communication (IPC).

PThreads Programming. Bradford Nichols et al. O'Reilly Media, 1996. A deeper reference to the POSIX threading API, Pthreads, supplementing this book.

Managing Projects with GNU Make. Robert Mecklenburg. O'Reilly Media, 2004. An excellent treatment on GNU Make, the classic tool for building software projects on Linux.

Version Control with Subversion. Ben Collins-Sussman et al. O'Reilly Media, 2004. A comprehensive take on Subversion, the successor of CVS for revision control and source code management on Unix systems, by three of Subversion's own authors.

Version Control with Git. Jon Loeliger et al. O'Reilly Media, 2012. A excellent introduction to Git, the sometimes confusing but always powerful distributed revision control system.

GDB Pocket Reference. Arnold Robbins. O'Reilly Media, 2005. A handy pocket guide to *gdb*, Linux's debugger.

Linux in a Nutshell. Ellen Siever et al. O'Reilly Media, 2009. A whirlwind reference to all things Linux, including many of the tools comprising Linux's development environment.

Books on the Linux Kernel

The two titles listed here cover the Linux kernel. Reasons for investigating this topic are threefold. First, the kernel provides the system call interface to user space and is thus the core of system programming. Second, the behaviors and idiosyncrasies of a kernel shed light on its interactions with the applications it runs. Finally, the Linux kernel is a wonderful chunk of code, and these books are fun.

Linux Kernel Development. Robert Love. Addison-Wesley, 2010. My own effort in this category is ideally suited to system programmers who want to know about the design and implementation of the Linux kernel. Not an API reference, this book offers a great discussion of the algorithms used and decisions made by the Linux kernel.

Linux Device Drivers. Jonathan Corbet et al. O'Reilly Media, 2005. This is a great guide to writing device drivers for the Linux kernel, with excellent API references. Although aimed at device drivers, the discussions will benefit programmers of any persuasion, including system programmers merely seeking more insight into the machinations of the Linux kernel. A great complement to my own Linux kernel book.

Books on Operating System Design

These two works, not specific to Linux, address operating system design in the abstract. As I've stressed in this book, a strong understanding of the system on which you develop can only improve your output.

Operating System Concepts. Abraham Silberschatz et al. Prentice Hall, 2012. An excellent introduction to operating systems, their history, and their underlying algorithms. Includes an excellent set of case studies.

UNIX Systems for Modern Architectures. Curt Schimmel. Addison-Wesley, 1994. This book, less on Unix than modern processor and cache architectures, is an excellent introduction to how operating systems cope with the complexities of modern systems. Although growing a bit dated, I still highly recommend it.