9.1 POSIX Times

POSIX specifies that systems should keep time in terms of seconds since the Epoch and that each day be accounted for by exactly 86,400 seconds. The *Epoch* is defined as 00:00 (midnight), January 1, 1970, Coordinated Universal Time (also called UTC, Greenwich Mean Time or GMT). POSIX does not specify how an implementation should align its system time with the actual time and date.

Most operations need to be measured with timers with greater than one-second resolution. Two POSIX extensions, the POSIX:XSI Extension and the POSIX:TMR Extension, define time resolutions of microseconds and nanoseconds, respectively.

9.1.1 Expressing time in seconds since the Epoch

The POSIX base standard supports only a time resolution of seconds and expresses time since the Epoch using a `time_t` type, which is usually implemented as a `long`. A program can access the system time (expressed in seconds since the Epoch) by calling the `time` function. If `tloc` is not `NULL`, the `time` function also stores the time in `*tloc`.

SYNOPSIS

```
#include <time.h>

time_t time(time_t *tloc);
```

POSIX:CX

If successful, `time` returns the number of seconds since the Epoch. If unsuccessful, `time` returns `(time_t)`–1. POSIX does not define any mandatory errors for `time`.

Exercise 9.1

The `time_t` type is usually implemented as a `long`. If a `long` is 32 bits, at approximately what date would `time_t` overflow? (Remember that one bit is used for the sign.) What date would cause an overflow if an `unsigned long` were used? What date would cause an overflow if a 64-bit data type were used?

Answer:

For a 32-bit `long`, time would overflow in approximately 68 years from January 1, 1970, so the system would not have a "Y2K" problem until the year 2038. For a `time_t` value that is an `unsigned long`, the overflow would occur in the year 2106, but this would not allow `time` to return an error. For a 64-bit data type, the overflow would not occur for another 292 billion years, long after the sun has died!

The `difftime` function computes the difference between two calendar times of type `time_t`, making it convenient for calculations involving time. The `difftime` function

has two `time_t` parameters and returns a `double` containing the first parameter minus the second.

SYNOPSIS

```
#include <time.h>

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

POSIX:CX

No errors are defined for `difftime`.

Example 9.2

The following program calculates the wall-clock time that it takes to execute `function_to_time`.

```
#include <stdio.h>
#include <time.h>
void function_to_time(void);

int main(void) {
   time_t tstart;

   tstart = time(NULL);
   function_to_time();
   printf("function_to_time took %f seconds of elapsed time\n",
           difftime(time(NULL), tstart));
   return 0;
}
```

simpletiming.c

Example 9.2 uses a time resolution of one second, which may not be accurate enough unless `function_to_time` involves substantial computation or waiting. Also, the `time` function measures wall-clock or elapsed time, which may not meaningfully reflect the amount of CPU time used. Section 9.1.5 presents alternative methods of timing code.

9.1.2 Displaying date and time

The `time_t` type is convenient for calculations requiring the difference between times, but it is cumbersome for printing dates. Also, a program should adjust dates and times to account for factors such as time zone, daylight-saving time and leap seconds.

The `localtime` function takes a parameter specifying the seconds since the Epoch and returns a structure with the components of the time (such as day, month and year) adjusted for local requirements. The `asctime` function converts the structure returned by `localtime` to a string. The `ctime` function is equivalent to `asctime(localtime(clock))`. The `gmtime` function takes a parameter representing seconds since the Epoch and returns a structure with the components of time expressed as Coordinated Universal Time (UTC).

SYNOPSIS

```
#include <time.h>

char *asctime(const struct tm *timeptr);
char *ctime(const time_t *clock);
struct tm *gmtime(const time_t *timer);
struct tm *localtime(const time_t *timer);
```

POSIX:CX

No errors are defined for these functions.

The `ctime` function takes one parameter, a pointer to a variable of type `time_t`, and returns a pointer to a 26-character English-language string. The `ctime` function takes into account both the time zone and daylight saving time. Each of the fields in the string has a constant width. The string might be stored as follows.

```
Sun Oct 06 02:21:35 1986\n\0
```

■ Example 9.3

The following program prints the date and time. The `printf` format did not include `'\n'` because `ctime` returns a string that ends in a newline.

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

int main(void) {
   time_t tcurrent;

   tcurrent = time(NULL);
   printf("The current time is %s", ctime(&tcurrent));
   return 0;
}
```

timeprint.c

□ Exercise 9.4

What is wrong with the following program that prints the time before and after the function `function_to_time` executes?

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

void function_to_time(void);

int main(void) {
   time_t tend, tstart;

   tstart = time(NULL);
   function_to_time();
   tend = time(NULL);
   printf("The time before was %sThe time after  was %s",
           ctime(&tstart), ctime(&tend));
   return 0;
}
```

badtiming.c

Answer:
The `ctime` function uses static storage to hold the time string. Both calls to `ctime` store the string in the same place, so the second call may overwrite the first value before it is used. Most likely, both times will be printed as the same value.

The `gmtime` and `localtime` functions break the time into separate fields to make it easy for programs to output components of the date or time. ISO C defines the `struct tm` structure to have the following members.

```
int tm_sec;       /* seconds after the minute [0,60] */
int tm_min;       /* minutes after the hour [0,59] */
int tm_hour;      /* hours since midnight [0,23] */
int tm_mday;      /* day of the month [1,31] */
int tm_mon;       /* months since January [0,11] */
int tm_year;      /* years since 1900 */
int tm_wday;      /* days since Sunday [0,6] */
int tm_yday;      /* days since January 1 [0,365] */
int tm_isdst;     /* flag indicating daylight-saving time */
```

■ Example 9.5

The following code segment prints the number of days since the beginning of the year.

```
struct tm *tcurrent;
time_t current_time;

current_time = time(NULL);
tcurrent = localtime(&current_time);
printf("%d days have elapsed since Jan 1\n", tcurrent->tm_yday);
```

Unfortunately, the `asctime`, `ctime` and `localtime` are not thread-safe. The POSIX:TSF Thread Safe Extension specifies thread-safe alternatives that have a caller-supplied buffer as an additional parameter.

SYNOPSIS

```
#include <time.h>

char *asctime_r(const struct tm *restrict timeptr, char *restrict buf);
char *ctime_r(const time_t *clock, char *buf);
struct tm *gmtime_r(const time_t *restrict timer,
                    struct tm *restrict result);
struct tm *localtime_r(const time_t *restrict timer,
                    struct tm *restrict result);
```

POSIX:TSF

If successful, these functions return a pointer to the parameter holding the result. For `asctime_r` and `ctime_r`, the result is in `buf`. For `gmtime_r` and `localtime_r`, the result is in `result`. If unsuccessful, these functions return a `NULL` pointer.