6

530128

System Data Files and Information

6.1 Introduction

A UNIX system requires numerous data files for normal operation: the password file `/etc/passwd` and the group file `/etc/group` are two files that are frequently used by various programs. For example, the password file is used every time a user logs in to a UNIX system and every time someone executes an `ls -l` command.

Historically, these data files have been ASCII text files and were read with the standard I/O library. But for larger systems, a sequential scan through the password file becomes time consuming. We want to be able to store these data files in a format other than ASCII text, but still provide an interface for an application program that works with any file format. The portable interfaces to these data files are the subject of this chapter. We also cover the system identification functions and the time and date functions.

6.2 Password File

The UNIX System's password file, called the user database by POSIX.1, contains the fields shown in Figure 6.1. These fields are contained in a `passwd` structure that is defined in `<pwd.h>`.

> Note that POSIX.1 specifies only 5 of the 10 fields in the `passwd` structure. Most platforms support at least 7 of the fields. The BSD-derived platforms support all 10.

Description	struct passwd member	POSIX.1	FreeBSD 8.0	Linux 3.2.0	Mac OS X 10.6.8	Solaris 10
user name	char *pw_name	•	•	•	•	•
encrypted password	char *pw_passwd		•	•	•	•
numerical user ID	uid_t pw_uid	•	•	•	•	•
numerical group ID	gid_t pw_gid	•	•	•	•	•
comment field	char *pw_gecos		•	•	•	•
initial working directory	char *pw_dir	•	•	•	•	•
initial shell (user program)	char *pw_shell	•	•	•	•	•
user access class	char *pw_class		•		•	
next time to change password	time_t pw_change		•		•	
account expiration time	time_t pw_expire		•		•	

Figure 6.1 Fields in /etc/passwd file

Historically, the password file has been stored in /etc/passwd and has been an ASCII file. Each line contains the fields described in Figure 6.1, separated by colons. For example, four lines from the /etc/passwd file on Linux could be

```
root:x:0:0:root:/root:/bin/bash
squid:x:23:23::/var/spool/squid:/dev/null
nobody:x:65534:65534:Nobody:/home:/bin/sh
sar:x:205:105:Stephen Rago:/home/sar:/bin/bash
```

Note the following points about these entries.

- There is usually an entry with the user name root. This entry has a user ID of 0 (the superuser).
- The encrypted password field contains a single character as a placeholder where older versions of the UNIX System used to store the encrypted password. Because it is a security hole to store the encrypted password in a file that is readable by everyone, encrypted passwords are now kept elsewhere. We'll cover this issue in more detail in the next section when we discuss passwords.
- Some fields in a password file entry can be empty. If the encrypted password field is empty, it usually means that the user does not have a password. (This is not recommended.) The entry for squid has one blank field: the comment field. An empty comment field has no effect.
- The shell field contains the name of the executable program to be used as the login shell for the user. The default value for an empty shell field is usually /bin/sh. Note, however, that the entry for squid has /dev/null as the login shell. Obviously, this is a device and cannot be executed, so its use here is to prevent anyone from logging in to our system as user squid.

  Many services have separate user IDs for the daemon processes (Chapter 13) that help implement the service. The squid entry is for the processes implementing the squid proxy cache service.

- There are several alternatives to using `/dev/null` to prevent a particular user from logging in to a system. For example, `/bin/false` is often used as the login shell. It simply exits with an unsuccessful (nonzero) status; the shell evaluates the exit status as false. It is also common to see `/bin/true` used to disable an account; it simply exits with a successful (zero) status. Some systems provide the `nologin` command, which prints a customizable error message and exits with a nonzero exit status.
- The `nobody` user name can be used to allow people to log in to a system, but with a user ID (65534) and group ID (65534) that provide no privileges. The only files that this user ID and group ID can access are those that are readable or writable by the world. (This approach assumes that there are no files specifically owned by user ID 65534 or group ID 65534, which should be the case.)
- Some systems that provide the `finger`(1) command support additional information in the comment field. Each of these fields is separated by a comma: the user's name, office location, office phone number, and home phone number. Additionally, an ampersand in the comment field is replaced with the login name (capitalized) by some utilities. For example, we could have

```
sar:x:205:105:Steve Rago, SF 5-121, 555-1111, 555-2222:/home/sar:/bin/sh
```

Then we could use `finger` to print information about Steve Rago.

```
$ finger -p sar
Login: sar                              Name: Steve Rago
Directory: /home/sar                    Shell: /bin/sh
Office:  SF 5-121,  555-1111            Home Phone:  555-2222
On since Mon Jan 19 03:57 (EST) on ttyv0 (messages off)
No Mail.
```

Even if your system doesn't support the `finger` command, these fields can still go into the comment field, since that field is simply a comment and not interpreted by system utilities.

Some systems provide the `vipw` command to allow administrators to edit the password file. The `vipw` command serializes changes to the password file and makes sure that any additional files are consistent with the changes made. It is also common for systems to provide similar functionality through graphical user interfaces.

POSIX.1 defines two functions to fetch entries from the password file. These functions allow us to look up an entry given a user's login name or numerical user ID.

```
#include <pwd.h>

struct passwd *getpwuid(uid_t uid);

struct passwd *getpwnam(const char *name);
```

Both return: pointer if OK, NULL on error

The `getpwuid` function is used by the `ls`(1) program to map the numerical user ID contained in an i-node into a user's login name. The `getpwnam` function is used by the `login`(1) program when we enter our login name.

Both functions return a pointer to a `passwd` structure that the functions fill in. This structure is usually a `static` variable within the function, so its contents are overwritten each time we call either of these functions.

These two POSIX.1 functions are fine if we want to look up either a login name or a user ID, but some programs need to go through the entire password file. Three functions can be used for this purpose: `getpwent`, `setpwent`, and `endpwent`.

```
#include <pwd.h>

struct passwd *getpwent(void);
                                    Returns: pointer if OK, NULL on error or end of file

void setpwent(void);

void endpwent(void);
```

These three functions are not part of the base POSIX.1 standard. They are defined as part of the XSI option in the Single UNIX Specification. As such, all UNIX systems are expected to provide them.

We call `getpwent` to return the next entry in the password file. As with the two POSIX.1 functions, `getpwent` returns a pointer to a structure that it has filled in. This structure is normally overwritten each time we call this function. If this is the first call to this function, it opens whatever files it uses. There is no order implied when we use this function; the entries can be in any order, because some systems use a hashed version of the file `/etc/passwd`.

The function `setpwent` rewinds whatever files it uses, and `endpwent` closes these files. When using `getpwent`, we must always be sure to close these files by calling `endpwent` when we're through. Although `getpwent` is smart enough to know when it has to open its files (the first time we call it), it never knows when we're through.

Example

Figure 6.2 shows an implementation of the function `getpwnam`.

```
#include <pwd.h>
#include <stddef.h>
#include <string.h>

struct passwd *
getpwnam(const char *name)
{
    struct passwd  *ptr;

    setpwent();
    while ((ptr = getpwent()) != NULL)
        if (strcmp(name, ptr->pw_name) == 0)
            break;      /* found a match */
    endpwent();
    return(ptr);    /* ptr is NULL if no match found */
}
```

Figure 6.2 The `getpwnam` function

The call to `setpwent` at the beginning of this function is self-defense: we ensure that the files are rewound, in case the caller has already opened them by calling `getpwent`. We call `endpwent` when we're done, because neither `getpwnam` nor `getpwuid` should leave any of the files open. □

6.3 Shadow Passwords

The encrypted password is a copy of the user's password that has been put through a one-way encryption algorithm. Because this algorithm is one-way, we can't guess the original password from the encrypted version.

Historically, the algorithm used always generated 13 printable characters from the 64-character set `[a-zA-Z0-9./]` (see Morris and Thompson [1979]). Some newer systems use alternative algorithms, such as MD5 or SHA-1, to generate longer encrypted password strings. (The more characters used to store the encrypted password, the more combinations there are, and the harder it will be to guess the password by trying all possible variations.) When we place a single character in the encrypted password field, we ensure that an encrypted password will never match this value.

Given an encrypted password, we can't apply an algorithm that inverts it and returns the plaintext password. (The plaintext password is what we enter at the `Password:` prompt.) But we could guess a password, run it through the one-way algorithm, and compare the result to the encrypted password. If user passwords were randomly chosen, this brute-force approach wouldn't be too successful. Users, however, tend to choose nonrandom passwords, such as spouse's name, street names, or pet names. A common experiment is for someone to obtain a copy of the password file and try guessing the passwords. (Chapter 4 of Garfinkel et al. [2003] contains additional details and history on passwords and the password encryption scheme used on UNIX systems.)

To make it more difficult to obtain the raw materials (the encrypted passwords), systems now store the encrypted password in another file, often called the *shadow password file*. Minimally, this file has to contain the user name and the encrypted password. Other information relating to the password is also stored here (Figure 6.3).

Description	`struct spwd` member
user login name	`char *sp_namp`
encrypted password	`char *sp_pwdp`
days since Epoch of last password change	`int sp_lstchg`
days until change allowed	`int sp_min`
days before change required	`int sp_max`
days warning for expiration	`int sp_warn`
days before account inactive	`int sp_inact`
days since Epoch when account expires	`int sp_expire`
reserved	`unsigned int sp_flag`

Figure 6.3 Fields in `/etc/shadow` file

The only two mandatory fields are the user's login name and encrypted password. The other fields control how often the password is to change—known as "password aging"—and how long an account is allowed to remain active.

The shadow password file should not be readable by the world. Only a few programs need to access encrypted passwords—`login`(1) and `passwd`(1), for example—and these programs are often set-user-ID root. With shadow passwords, the regular password file, `/etc/passwd`, can be left readable by the world.

On Linux 3.2.0 and Solaris 10, a separate set of functions is available to access the shadow password file, similar to the set of functions used to access the password file.

```
#include <shadow.h>

struct spwd *getspnam(const char *name);

struct spwd *getspent(void);
                                        Both return: pointer if OK, NULL on error

void setspent(void);

void endspent(void);
```

On FreeBSD 8.0 and Mac OS X 10.6.8, there is no shadow password structure. The additional account information is stored in the password file (refer back to Figure 6.1).

6.4 Group File

The UNIX System's group file, called the group database by POSIX.1, contains the fields shown in Figure 6.4. These fields are contained in a `group` structure that is defined in `<grp.h>`.

Description	`struct group` member	POSIX.1	FreeBSD 8.0	Linux 3.2.0	Mac OS X 10.6.8	Solaris 10
group name	`char *gr_name`	•	•	•	•	•
encrypted password	`char *gr_passwd`		•	•	•	•
numerical group ID	`int gr_gid`	•	•	•	•	•
array of pointers to individual user names	`char **gr_mem`	•	•	•	•	•

Figure 6.4 Fields in `/etc/group` file

The field `gr_mem` is an array of pointers to the user names that belong to this group. This array is terminated by a null pointer.

We can look up either a group name or a numerical group ID with the following two functions, which are defined by POSIX.1.

```
#include <grp.h>

struct group *getgrgid(gid_t gid);

struct group *getgrnam(const char *name);
                                        Both return: pointer if OK, NULL on error
```

Like the password file functions, both of these functions normally return pointers to a `static` variable, which is overwritten on each call.

If we want to search the entire group file, we need some additional functions. The following three functions are like their counterparts for the password file.

```
#include <grp.h>

struct group *getgrent(void);
```

Returns: pointer if OK, `NULL` on error or end of file

```
void setgrent(void);

void endgrent(void);
```

These three functions are not part of the base POSIX.1 standard. They are defined as part of the XSI option in the Single UNIX Specification. All UNIX Systems provide them.

The `setgrent` function opens the group file, if it's not already open, and rewinds it. The `getgrent` function reads the next entry from the group file, opening the file first, if it's not already open. The `endgrent` function closes the group file.

6.5 Supplementary Group IDs

The use of groups in the UNIX System has changed over time. With Version 7, each user belonged to a single group at any point in time. When we logged in, we were assigned the real group ID corresponding to the numerical group ID in our password file entry. We could change this at any point by executing `newgrp`(1). If the `newgrp` command succeeded (refer to the manual page for the permission rules), our real group ID was changed to the new group's ID, and this value was used for all subsequent file access permission checks. We could always go back to our original group by executing `newgrp` without any arguments.

This form of group membership persisted until it was changed in 4.2BSD (circa 1983). With 4.2BSD, the concept of supplementary group IDs was introduced. Not only did we belong to the group corresponding to the group ID in our password file entry, but we could also belong to as many as 16 additional groups. The file access permission checks were modified so that in addition to comparing the file's group ID to the process effective group ID, it was also compared to all the supplementary group IDs.

Supplementary group IDs are a required feature of POSIX.1. (In older versions of POSIX.1, they were optional.) The constant `NGROUPS_MAX` (Figure 2.11) specifies the number of supplementary group IDs. A common value is 16 (Figure 2.15).

The advantage of using supplementary group IDs is that we no longer have to change groups explicitly. It is not uncommon to belong to multiple groups (i.e., participate in multiple projects) at the same time.

Three functions are provided to fetch and set the supplementary group IDs.

```
#include <unistd.h>

int getgroups(int gidsetsize, gid_t grouplist[]);
```

Returns: number of supplementary group IDs if OK, –1 on error

```
#include <grp.h>     /* on Linux */
#include <unistd.h> /* on FreeBSD, Mac OS X, and Solaris */

int setgroups(int ngroups, const gid_t grouplist[]);

#include <grp.h>     /* on Linux and Solaris */
#include <unistd.h> /* on FreeBSD and Mac OS X */

int initgroups(const char *username, gid_t basegid);
```

Both return: 0 if OK, –1 on error

Of these three functions, only `getgroups` is specified by POSIX.1. Because `setgroups` and `initgroups` are privileged operations, they are not part of POSIX.1. All four platforms covered in this book support all three functions, but on Mac OS X 10.6.8, *basegid* is declared to be of type `int`.

The `getgroups` function fills in the array *grouplist* with the supplementary group IDs. Up to *gidsetsize* elements are stored in the array. The number of supplementary group IDs stored in the array is returned by the function.

As a special case, if *gidsetsize* is 0, the function returns only the number of supplementary group IDs. The array *grouplist* is not modified. (This allows the caller to determine the size of the *grouplist* array to allocate.)

The `setgroups` function can be called by the superuser to set the supplementary group ID list for the calling process: *grouplist* contains the array of group IDs, and *ngroups* specifies the number of elements in the array. The value of *ngroups* cannot be larger than `NGROUPS_MAX`.

The `setgroups` function is usually called from the `initgroups` function, which reads the entire group file—with the functions `getgrent`, `setgrent`, and `endgrent`, which we described earlier—and determines the group membership for *username*. It then calls `setgroups` to initialize the supplementary group ID list for the user. One must be superuser to call `initgroups`, since it calls `setgroups`. In addition to finding all the groups that *username* is a member of in the group file, `initgroups` includes *basegid* in the supplementary group ID list; *basegid* is the group ID from the password file for *username*.

The `initgroups` function is called by only a few programs. The `login`(1) program, for example, calls it when we log in.

6.6 Implementation Differences

We've already discussed the shadow password file supported by Linux and Solaris. FreeBSD and Mac OS X store encrypted passwords differently. Figure 6.5 summarizes how the four platforms covered in this book store user and group information.

Information	FreeBSD 8.0	Linux 3.2.0	Mac OS X 10.6.8	Solaris 10
account information	`/etc/passwd`	`/etc/passwd`	Directory Services	`/etc/passwd`
encrypted passwords	`/etc/master.passwd`	`/etc/shadow`	Directory Services	`/etc/shadow`
hashed password files?	yes	no	no	no
group information	`/etc/group`	`/etc/group`	Directory Services	`/etc/group`

Figure 6.5 Account implementation differences

On FreeBSD, the shadow password file is `/etc/master.passwd`. Special commands are used to edit it, which in turn generate a copy of `/etc/passwd` from the shadow password file. In addition, hashed versions of the files are generated: `/etc/pwd.db` is the hashed version of `/etc/passwd`, and `/etc/spwd.db` is the hashed version of `/etc/master.passwd`. These provide better performance for large installations.

On Mac OS X, however, `/etc/passwd` and `/etc/master.passwd` are used only in single-user mode (when the system is undergoing maintenance; single-user mode usually means that no system services are enabled). In multiuser mode—during normal operation—the Directory Services daemon provides access to account information for users and groups.

Although Linux and Solaris support similar shadow password interfaces, there are some subtle differences. For example, the integer fields shown in Figure 6.3 are defined as type `int` on Solaris, but as `long int` on Linux. Another difference is the account-inactive field: Solaris defines it to be the number of days since the user last logged in to the system after which the account will be automatically disabled, whereas Linux defines it to be the number of days after the maximum password age has been reached during which the password will still be accepted.

On many systems, the user and group databases are implemented using the Network Information Service (NIS). This allows administrators to edit a master copy of the databases and distribute them automatically to all servers in an organization. Client systems contact servers to look up information about users and groups. NIS+ and the Lightweight Directory Access Protocol (LDAP) provide similar functionality. Many systems control the method used to administer each type of information through the `/etc/nsswitch.conf` configuration file.

6.7 Other Data Files

We've discussed only two of the system's data files so far: the password file and the group file. Numerous other files are used by UNIX systems in normal day-to-day operation. For example, the BSD networking software has one data file for the services provided by the various network servers (`/etc/services`), one for the protocols (`/etc/protocols`), and one for the networks (`/etc/networks`). Fortunately, the interfaces to these various files are like the ones we've already described for the password and group files.

The general principle is that every data file has at least three functions:

1. A `get` function that reads the next record, opening the file if necessary. These functions normally return a pointer to a structure. A null pointer is returned when the end of file is reached. Most of the `get` functions return a pointer to a `static` structure, so we always have to copy the structure if we want to save it.
2. A `set` function that opens the file, if not already open, and rewinds the file. We use this function when we know we want to start again at the beginning of the file.
3. An `end` entry that closes the data file. As we mentioned earlier, we always have to call this function when we're done, to close all the files.

Additionally, if the data file supports some form of keyed lookup, routines are provided to search for a record with a specific key. For example, two keyed lookup routines are provided for the password file: `getpwnam` looks for a record with a specific user name, and `getpwuid` looks for a record with a specific user ID.

Figure 6.6 shows some of these routines, which are common to UNIX systems. In this figure, we show the functions for the password files and group file, which we discussed earlier in this chapter, and some of the networking functions. There are `get`, `set`, and `end` functions for all the data files in this figure.

Description	Data file	Header	Structure	Additional keyed lookup functions
passwords	`/etc/passwd`	`<pwd.h>`	`passwd`	`getpwnam`, `getpwuid`
groups	`/etc/group`	`<grp.h>`	`group`	`getgrnam`, `getgrgid`
shadow	`/etc/shadow`	`<shadow.h>`	`spwd`	`getspnam`
hosts	`/etc/hosts`	`<netdb.h>`	`hostent`	`getnameinfo`, `getaddrinfo`
networks	`/etc/networks`	`<netdb.h>`	`netent`	`getnetbyname`, `getnetbyaddr`
protocols	`/etc/protocols`	`<netdb.h>`	`protoent`	`getprotobyname`, `getprotobynumber`
services	`/etc/services`	`<netdb.h>`	`servent`	`getservbyname`, `getservbyport`

Figure 6.6 Similar routines for accessing system data files

Under Solaris, the last four data files in Figure 6.6 are symbolic links to files of the same name in the directory `/etc/inet`. Most UNIX System implementations have additional functions that are like these, but the additional functions tend to deal with system administration files and are specific to each implementation.

6.8 Login Accounting

Two data files provided with most UNIX systems are the `utmp` file, which keeps track of all the users currently logged in, and the `wtmp` file, which keeps track of all logins and logouts. With Version 7, one type of record was written to both files, a binary record consisting of the following structure:

```
struct utmp {
  char  ut_line[8]; /* tty line: "ttyh0", "ttyd0", "ttyp0", ... */
  char  ut_name[8]; /* login name */
  long  ut_time;    /* seconds since Epoch */
};
```

On login, one of these structures was filled in and written to the `utmp` file by the `login` program, and the same structure was appended to the `wtmp` file. On logout, the entry in the `utmp` file was erased—filled with null bytes—by the `init` process, and a new entry was appended to the `wtmp` file. This logout entry in the `wtmp` file had the `ut_name` field zeroed out. Special entries were appended to the `wtmp` file to indicate when the system was rebooted and right before and after the system's time and date was changed. The who(1) program read the `utmp` file and printed its contents in a readable form. Later versions of the UNIX System provided the `last`(1) command, which read through the `wtmp` file and printed selected entries.

Most versions of the UNIX System still provide the `utmp` and `wtmp` files, but as expected, the amount of information in these files has grown. The 20-byte structure that was written by Version 7 grew to 36 bytes with SVR2, and the extended `utmp` structure with SVR4 takes more than 350 bytes!

> The detailed format of these records in Solaris is given in the `utmpx`(4) manual page. With Solaris 10, both files are in the `/var/adm` directory. Solaris provides numerous functions described in `getutxent`(3) to read and write these two files.
>
> On FreeBSD 8.0 and Linux 3.2.0, the `utmp`(5) manual page gives the format of their versions of these login records. The pathnames of these two files are `/var/run/utmp` and `/var/log/wtmp`. On Mac OS X 10.6.8, the `utmp` and `wtmp` files do not exist. As of Mac OS X 10.5, the information found in the `wtmp` file can be obtained from the system logging facility, and the `utmpx` file contains information about the active login sessions.

6.9 System Identification

POSIX.1 defines the `uname` function to return information on the current host and operating system.

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

int uname(struct utsname *name);
```

Returns: non-negative value if OK, –1 on error

We pass the address of a `utsname` structure to this function, and the function then fills it in. POSIX.1 defines only the minimum fields in the structure, which are all character arrays, and it's up to each implementation to set the size of each array. Some implementations provide additional fields in the structure.

```
struct utsname {
  char  sysname[];    /* name of the operating system */
  char  nodename[];   /* name of this node */
  char  release[];    /* current release of operating system */
  char  version[];    /* current version of this release */
  char  machine[];    /* name of hardware type */
};
```

Each string is null terminated. The maximum name lengths, including the terminating null byte, supported by the four platforms discussed in this book are listed in Figure 6.7. The information in the `utsname` structure can usually be printed with the `uname`(1) command.

> POSIX.1 warns that the `nodename` element may not be adequate to reference the host on a communications network. This function is from System V, and in older days, the `nodename` element was adequate for referencing the host on a UUCP network.
>
> Realize also that the information in this structure does not give any information on the POSIX.1 level. This should be obtained using `_POSIX_VERSION`, as described in Section 2.6.
>
> Finally, this function gives us a way only to fetch the information in the structure; there is nothing specified by POSIX.1 about initializing this information.

Historically, BSD-derived systems provided the `gethostname` function to return only the name of the host. This name is usually the name of the host on a TCP/IP network.

```
#include <unistd.h>

int gethostname(char *name, int namelen);
```

Returns: 0 if OK, –1 on error

The *namelen* argument specifies the size of the *name* buffer. If enough space is provided, the string returned through *name* is null terminated. If insufficient room is provided, however, it is unspecified whether the string is null terminated.

The `gethostname` function, which is now defined as part of POSIX.1, specifies that the maximum host name length is `HOST_NAME_MAX`. Figure 6.7 summarizes the maximum name lengths supported by the four implementations covered in this book.

Interface	Maximum name length			
	FreeBSD 8.0	Linux 3.2.0	Mac OS X 10.6.8	Solaris 10
`uname`	256	65	256	257
`gethostname`	256	64	256	256

Figure 6.7 System identification name limits

If the host is connected to a TCP/IP network, the host name is normally the fully qualified domain name of the host.

There is also a `hostname`(1) command that can fetch or set the host name. (The host name is set by the superuser using a similar function, `sethostname`.) The host name is normally set at bootstrap time from one of the start-up files invoked by `/etc/rc` or `init`.

6.10 Time and Date Routines

The basic time service provided by the UNIX kernel counts the number of seconds that have passed since the Epoch: 00:00:00 January 1, 1970, Coordinated Universal Time (UTC). In Section 1.10, we said that these seconds are represented in a `time_t` data type, and we call them *calendar times*. These calendar times represent both the time and the date. The UNIX System has always differed from other operating systems in (a) keeping time in UTC instead of the local time, (b) automatically handling conversions, such as daylight saving time, and (c) keeping the time and date as a single quantity.

The `time` function returns the current time and date.

```
#include <time.h>

time_t time(time_t *calptr);
```

Returns: value of time if OK, –1 on error

The time value is always returned as the value of the function. If the argument is non-null, the time value is also stored at the location pointed to by *calptr*.

The real-time extensions to POSIX.1 added support for multiple system clocks. In Version 4 of the Single UNIX Specification, the interfaces used to control these clocks were moved from an option group to the base. A clock is identified by the `clockid_t` type. Standard values are summarized in Figure 6.8.

Identifier	Option	Description
CLOCK_REALTIME		real system time
CLOCK_MONOTONIC	_POSIX_MONOTONIC_CLOCK	real system time with no negative jumps
CLOCK_PROCESS_CPUTIME_ID	_POSIX_CPUTIME	CPU time for calling process
CLOCK_THREAD_CPUTIME_ID	_POSIX_THREAD_CPUTIME	CPU time for calling thread

Figure 6.8 Clock type identifiers

The `clock_gettime` function can be used to get the time of the specified clock. The time is returned in a `timespec` structure, introduced in Section 4.2, which expresses time values in terms of seconds and nanoseconds.

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

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

Returns: 0 if OK, –1 on error

When the clock ID is set to `CLOCK_REALTIME`, the `clock_gettime` function provides similar functionality to the `time` function, except with `clock_gettime`, we might be able to get a higher-resolution time value if the system supports it.

We can use the `clock_getres` function to determine the resolution of a given system clock.

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

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

Returns: 0 if OK, –1 on error

The `clock_getres` function initializes the `timespec` structure pointed to by the *tsp* argument to the resolution of the clock corresponding to the *clock_id* argument. For example, if the resolution is 1 millisecond, then the `tv_sec` field will contain 0 and the `tv_nsec` field will contain the value 1000000.

To set the time for a particular clock, we can call the `clock_settime` function.

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

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

Returns: 0 if OK, –1 on error

We need the appropriate privileges to change a clock's time. Some clocks, however, can't be modified.

> Historically, on implementations derived from System V, the `stime`(2) function was called to set the system time, whereas BSD-derived systems used `settimeofday`(2).

Version 4 of the Single UNIX Specification specifies that the `gettimeofday` function is now obsolescent. However, a lot of programs still use it, because it provides greater resolution (up to a microsecond) than the `time` function.

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

int gettimeofday(struct timeval *restrict tp, void *restrict tzp);
```

Returns: 0 always

The only legal value for *tzp* is `NULL`; other values result in unspecified behavior. Some platforms support the specification of a time zone through the use of *tzp*, but this is implementation specific and not defined by the Single UNIX Specification.

The `gettimeofday` function stores the current time as measured from the Epoch in the memory pointed to by *tp*. This time is represented as a `timeval` structure, which stores seconds and microseconds.

Once we have the integer value that counts the number of seconds since the Epoch, we normally call a function to convert it to a broken-down time structure, and then call another function to generate a human-readable time and date. Figure 6.9 shows the relationships between the various time functions. (The three functions in this figure that are shown with dashed lines—`localtime`, `mktime`, and `strftime`—are all affected by the `TZ` environment variable, which we describe later in this section. The dotted lines show how the calendar time is obtained from time-related structures.)

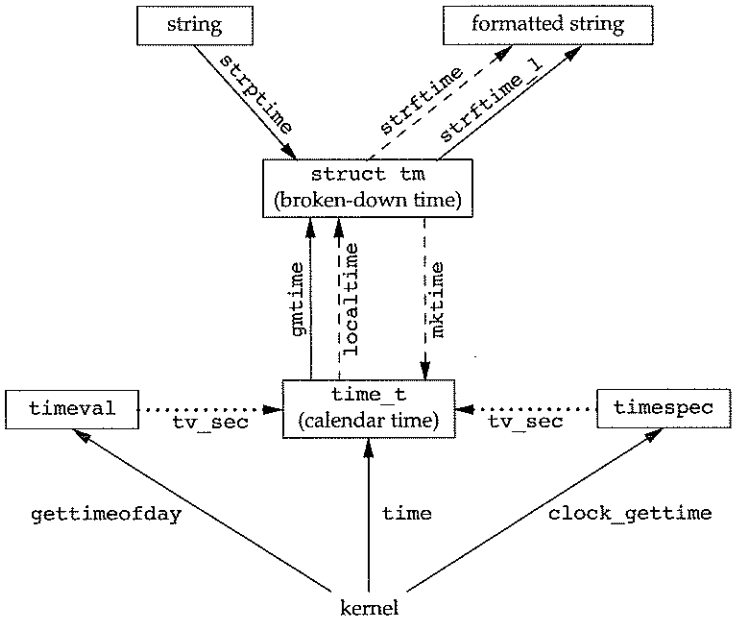

Figure 6.9 Relationship of the various time functions

The two functions `localtime` and `gmtime` convert a calendar time into what's called a broken-down time, a `tm` structure.

```
struct tm {      /* a broken-down time */
  int  tm_sec;   /* seconds after the minute: [0 - 60] */
  int  tm_min;   /* minutes after the hour: [0 - 59] */
  int  tm_hour;  /* hours after 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; /* daylight saving time flag: <0, 0, >0 */
};
```

The reason that the seconds can be greater than 59 is to allow for a leap second. Note that all the fields except the day of the month are 0-based. The daylight saving time flag is positive if daylight saving time is in effect, 0 if it's not in effect, and negative if the information isn't available.

> In older versions of the Single UNIX Specification, double leap seconds were allowed. Thus the valid range of values for the `tm_sec` member was 0–61. The formal definition of UTC doesn't allow for double leap seconds, so the valid range for seconds is now 0–60.

```
#include <time.h>

struct tm *gmtime(const time_t *calptr);

struct tm *localtime(const time_t *calptr);
```

Both return: pointer to broken-down time, NULL on error

The difference between `localtime` and `gmtime` is that the first converts the calendar time to the local time, taking into account the local time zone and daylight saving time flag, whereas the latter converts the calendar time into a broken-down time expressed as UTC.

The function `mktime` takes a broken-down time, expressed as a local time, and converts it into a `time_t` value.

```
#include <time.h>

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

Returns: calendar time if OK, –1 on error

The `strftime` function is a `printf`-like function for time values. It is complicated by the multitude of arguments available to customize the string it produces.

```
#include <time.h>

size_t strftime(char *restrict buf, size_t maxsize,
                const char *restrict format,
                const struct tm *restrict tmptr);

size_t strftime_l(char *restrict buf, size_t maxsize,
                const char *restrict format,
                const struct tm *restrict tmptr, locale_t locale);
```

Both return: number of characters stored in array if room, 0 otherwise

Two older functions, `asctime` and `ctime`, can be used to produce a 26-byte printable string similar to the default output of the `date`(1) command. However, these functions are now marked obsolescent, because they are susceptible to buffer overflow problems.

The `strftime` and `strftime_l` functions are the same, except that the `strftime_l` function allows the caller to specify the locale as an argument. The `strftime` function uses the locale specified by the locale environment variables (see Figure 7.7) and the timezone specified by the `TZ` environment variable.

The *tmptr* argument is the time value to format, specified by a pointer to a broken-down time value. The formatted result is stored in the array *buf* whose size is *maxsize* characters. If the size of the result, including the terminating null, fits in the buffer, these functions return the number of characters stored in *buf*, excluding the terminating null. Otherwise, these functions return 0.

The *format* argument controls the formatting of the time value. Like the `printf` functions, conversion specifiers are given as a percent sign followed by a special character. All other characters in the *format* string are copied to the output. Two percent signs in a row generate a single percent sign in the output. Unlike the `printf`

functions, each conversion specified generates a different fixed-size output string—there are no field widths in the *format* string. Figure 6.10 describes the 37 ISO C conversion specifiers.

Format	Description	Example
%a	abbreviated weekday name	Thu
%A	full weekday name	Thursday
%b	abbreviated month name	Jan
%B	full month name	January
%c	date and time	Thu Jan 19 21:24:52 2012
%C	year/100: [00–99]	20
%d	day of the month: [01–31]	19
%D	date [MM/DD/YY]	01/19/12
%e	day of month (single digit preceded by space) [1–31]	19
%F	ISO 8601 date format [YYYY–MM–DD]	2012-01-19
%g	last two digits of ISO 8601 week-based year [00–99]	12
%G	ISO 8601 week-based year	2012
%h	same as %b	Jan
%H	hour of the day (24-hour format): [00–23]	21
%I	hour of the day (12-hour format): [01–12]	09
%j	day of the year: [001–366]	019
%m	month: [01–12]	01
%M	minute: [00–59]	24
%n	newline character	
%p	AM/PM	PM
%r	locale's time (12-hour format)	09:24:52 PM
%R	same as %H:%M	21:24
%S	second: [00–60]	52
%t	horizontal tab character	
%T	same as %H:%M:%S	21:24:52
%u	ISO 8601 weekday [Monday = 1, 1–7]	4
%U	Sunday week number: [00–53]	03
%V	ISO 8601 week number: [01–53]	03
%w	weekday: [0 = Sunday, 0–6]	4
%W	Monday week number: [00–53]	03
%x	locale's date	01/19/12
%X	locale's time	21:24:52
%y	last two digits of year: [00–99]	12
%Y	year	2012
%z	offset from UTC in ISO 8601 format	-0500
%Z	time zone name	EST
%%	translates to a percent sign	%

Figure 6.10 Conversion specifiers for `strftime`

The third column of this figure is from the output of `strftime` under Mac OS X, corresponding to the time and date `Thu Jan 19 21:24:52 EST 2012`.

The only specifiers that are not self-evident are `%U`, `%V`, and `%W`. The `%U` specifier represents the week number of the year, where the week containing the first Sunday is week 1. The `%W` specifier represents the week number of the year, where the week containing the first Monday is week 1. The `%V` specifier is different. If the week

containing the first day in January has four or more days in the new year, then this is treated as week 1. Otherwise, it is treated as the last week of the previous year. In both cases, Monday is treated as the first day of the week.

As with `printf`, `strftime` supports modifiers for some of the conversion specifiers. The `E` and `O` modifiers can be used to generate an alternative format if one is supported by the locale.

Some systems support additional, nonstandard extensions to the *format* string for `strftime`.

Example

Figure 6.11 shows how to use several of the time functions discussed in this chapter. In particular, it shows how `strftime` can be used to print a string containing the current date and time.

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

int
main(void)
{
    time_t t;
    struct tm *tmp;
    char buf1[16];
    char buf2[64];

    time(&t);
    tmp = localtime(&t);
    if (strftime(buf1, 16, "time and date: %r, %a %b %d, %Y", tmp) == 0)
        printf("buffer length 16 is too small\n");
    else
        printf("%s\n", buf1);
    if (strftime(buf2, 64, "time and date: %r, %a %b %d, %Y", tmp) == 0)
        printf("buffer length 64 is too small\n");
    else
        printf("%s\n", buf2);
    exit(0);
}
```

Figure 6.11 Using the `strftime` function

Recall the relationship of the various time functions shown in Figure 6.9. Before we can print the time in a human-readable format, we need to get the time and convert it into a broken-down time structure. Sample output from Figure 6.11 is

```
$ ./a.out
buffer length 16 is too small
time and date: 11:12:35 PM, Thu Jan 19, 2012
```

□

The `strptime` function is the inverse of `strftime`. It takes a string and converts it into a broken-down time.

```
#include <time.h>

char *strptime(const char *restrict buf, const char *restrict format,
               struct tm *restrict tmptr);
```

Returns: pointer to one character past last character parsed, `NULL` otherwise

The *format* argument describes the format of the string in the buffer pointed to by the *buf* argument. The format specification is similar, although it differs slightly from the specification for the `strftime` function. The conversion specifiers for the `strptime` function are summarized in Figure 6.12.

Format	Description
`%a`	abbreviated or full weekday name
`%A`	same as `%a`
`%b`	abbreviated or full month name
`%B`	same as `%b`
`%c`	date and time
`%C`	all but the last two digits of the year
`%d`	day of the month: [01–31]
`%D`	date [MM/DD/YY]
`%e`	same as `%d`
`%h`	same as `%b`
`%H`	hour of the day (24-hour format): [00–23]
`%I`	hour of the day (12-hour format): [01–12]
`%j`	day of the year: [001–366]
`%m`	month: [01–12]
`%M`	minute: [00–59]
`%n`	any white space
`%p`	AM/PM
`%r`	locale's time (12-hour format, AM/PM notation)
`%R`	time as `%H:%M`
`%S`	second: [00–60]
`%t`	any white space
`%T`	time as `%H:%M:%S`
`%U`	Sunday week number: [00–53]
`%w`	weekday: [0 = Sunday, 0–6]
`%W`	Monday week number: [00–53]
`%x`	locale's date
`%X`	locale's time
`%y`	last two digits of year: [00–99]
`%Y`	year
`%%`	translates to a percent sign

Figure 6.12 Conversion specifiers for `strptime`

We mentioned that the three functions in Figure 6.9 with dashed lines were affected by the `TZ` environment variable: `localtime`, `mktime`, and `strftime`. If defined, the value of this environment variable is used by these functions instead of the default time

zone. If the variable is defined to be a null string, such as `TZ=`, then UTC is normally used. The value of this environment variable is often something like `TZ=EST5EDT`, but POSIX.1 allows a much more detailed specification. Refer to the Environment Variables chapter of the Single UNIX Specification [Open Group 2010] for all the details on the `TZ` variable.

More information on the `TZ` environment variable can be found in the `tzset`(3) manual page.

6.11 Summary

The password file and the group file are used on all UNIX systems. We've looked at the various functions that read these files. We've also talked about shadow passwords, which can enhance system security. Supplementary group IDs provide a way to participate in multiple groups at the same time. We also looked at how similar functions are provided by most systems to access other system-related data files. We discussed the POSIX.1 functions that programs can use to identify the system on which they are running. We finished the chapter by looking at the time and date functions provided by ISO C and the Single UNIX Specification.

Exercises

6.1 If the system uses a shadow file and we need to obtain the encrypted password, how do we do so?

6.2 If you have superuser access and your system uses shadow passwords, implement the previous exercise.

6.3 Write a program that calls `uname` and prints all the fields in the `utsname` structure. Compare the output to the output from the `uname`(1) command.

6.4 Calculate the latest time that can be represented by the `time_t` data type. After it wraps around, what happens?

6.5 Write a program to obtain the current time and print it using `strftime`, so that it looks like the default output from `date`(1). Set the `TZ` environment variable to different values and see what happens.