8

USERS AND GROUPS

Every user has a unique login name and an associated numeric user identifier (UID). Users can belong to one or more groups. Each group also has a unique name and a group identifier (GID).

The primary purpose of user and group IDs is to determine ownership of various system resources and to control the permissions granted to processes accessing those resources. For example, each file belongs to a particular user and group, and each process has a number of user and group IDs that determine who owns the process and what permissions it has when accessing a file (see Chapter 9 for details).

In this chapter, we look at the system files that are used to define the users and groups on the system, and then describe the library functions used to retrieve information from these files. We conclude with a discussion of the *crypt()* function, which is used to encrypt and authenticate login passwords.

8.1 The Password File: `/etc/passwd`

The system *password file*, `/etc/passwd`, contains one line for each user account on the system. Each line is composed of seven fields separated by colons (`:`), as in the following example:

```
mtk:x:1000:100:Michael Kerrisk:/home/mtk:/bin/bash
```

In order, these fields are as follows:

- *Login name*: This is the unique name that the user must enter in order to log in. Often, this is also called the username. We can also consider the login name to be the human-readable (symbolic) identifier corresponding to the numeric user identifier (described in a moment). Programs such as *ls(1)* display this name, rather than the numeric user ID associated with the file, when asked to show the ownership of a file (as in *ls –l*).
- *Encrypted password*: This field contains a 13-character encrypted password, which we describe in more detail in Section 8.5. If the password field contains any other string—in particular, a string of other than 13 characters—then logins to this account are disabled, since such a string can't represent a valid encrypted password. Note, however, that this field is ignored if shadow passwords have been enabled (which is typical). In this case, the password field in `/etc/passwd` conventionally contains the letter *x* (although any nonempty character string may appear), and the encrypted password is instead stored in the shadow password file (Section 8.2). If the password field in `/etc/passwd` is empty, then no password is required to log in to this account (this is true even if shadow passwords are enabled).

  > Here, we assume that passwords are encrypted using Data Encryption Standard (DES), the historical and still widely used UNIX password-encryption scheme. It is possible to replace DES with other schemes, such as MD5, which produces a 128-bit *message digest* (a kind of hash) of its input. This value is stored as a 34-character string in the password (or shadow password) file.

- *User ID* (UID): This is the numeric ID for this user. If this field has the value 0, then this account has superuser privileges. There is normally one such account, with the login name *root*. On Linux 2.2 and earlier, user IDs are maintained as 16-bit values, allowing the range 0 through to 65,535; on Linux 2.4 and later, they are stored using 32 bits, allowing a much larger range.

  > It is possible (but unusual) to have more than one record in the password file with the same user ID, thus permitting multiple login names for the same user ID. This allows multiple users to access the same resources (e.g., files) using different passwords. The different login names can be associated with different sets of group IDs.

- *Group ID* (GID): This is the numeric ID of the first of the groups of which this user is a member. Further group memberships for this user are defined in the system group file.
- *Comment*: This field holds text about the user. This text is displayed by various programs, such as *finger(1)*.
- *Home directory*: This is the initial directory into which the user is placed after logging in. This field becomes the value of the `HOME` environment variable.
- *Login shell*: This is the program to which control is transferred once the user is logged in. Usually, this is one of the shells, such as *bash*, but it can be any program. If this field is empty, then the login shell defaults to `/bin/sh`, the Bourne shell. This field becomes the value of the `SHELL` environment variable.

On a stand-alone system, all the password information resides in the file `/etc/passwd`. However, if we are using a system such as Network Information System (NIS) or Lightweight Directory Access Protocol (LDAP) to distribute passwords in a network environment, part or all of this information resides on a remote system. As long as programs accessing password information employ the functions described later in this chapter (*getpwnam()*, *getpwuid()*, and so on), the use of NIS or LDAP is transparent to applications. Similar comments apply regarding the shadow password and group files discussed in the following sections.

8.2 The Shadow Password File: `/etc/shadow`

Historically, UNIX systems maintained all user information, including the encrypted password, in `/etc/passwd`. This presented a security problem. Since various unprivileged system utilities needed to have read access to other information in the password file, it had to be made readable to all users. This opened the door for password-cracking programs, which try encrypting large lists of likely passwords (e.g., standard dictionary words or people's names) to see if they match the encrypted password of a user. The *shadow password file*, `/etc/shadow`, was devised as a method of preventing such attacks. The idea is that all of the nonsensitive user information resides in the publicly readable password file, while encrypted passwords are maintained in the shadow password file, which is readable only by privileged programs.

In addition to the login name, which provides the match to the corresponding record in the password file, and the encrypted password, the shadow password file also contains a number of other security-related fields. Further details on these fields can be found in the *shadow(5)* manual page. We'll concern ourselves mainly with the encrypted password field, which we discuss in greater detail when looking at the *crypt()* library function later in Section 8.5.

SUSv3 doesn't specify shadow passwords, and not all UNIX implementations provide this feature.

8.3 The Group File: `/etc/group`

For various administrative purposes, in particular, controlling access to files and other system resources, it is useful to organize users into *groups*.

The set of groups to which a user belongs is defined by the combination of the group ID field in the user's password entry and the groups under which the user is listed in the group file. This strange split of information across two files is historical in origin. In early UNIX implementations, a user could be a member of only one group at a time. A user's initial group membership at login was determined by the group ID field of the password file and could be changed thereafter using the *newgrp(1)* command, which required the user to supply the group password (if the group was password protected). 4.2BSD introduced the concept of multiple simultaneous group memberships, which was later standardized in POSIX.1-1990. Under this scheme, the group file listed the extra group memberships of each user. (The *groups(1)* command displays the groups of which the shell process is a member, or, if one or more usernames are supplied as command-line arguments, then the group memberships of those users.)

The *group file*, /etc/group, contains one line for each group in the system. Each line consists of four colon-separated fields, as in the following examples:

```
users:x:100:
jambit:x:106:claus,felli,frank,harti,markus,martin,mtk,paul
```

In order, these fields are as follows:

- *Group name*: This is the name of the group. Like the login name in the password file, we can consider this to be the human-readable (symbolic) identifier corresponding to the numeric group identifier.
- *Encrypted password*: This field contains an optional password for the group. With the advent of multiple group memberships, group passwords are nowadays rarely used on UNIX systems. Nevertheless, it is possible to place a password on a group (a privileged user can do this using the *passwd* command). If a user is not a member of the group, *newgrp(1)* requests this password before starting a new shell whose group memberships include that group. If password shadowing is enabled, then this field is ignored (in this case, conventionally it contains just the letter *x*, but any string, including an empty string, may appear) and the encrypted passwords are actually kept in the *shadow group file*, /etc/gshadow, which can be accessed only by privileged users and programs. Group passwords are encrypted in a similar fashion to user passwords (Section 8.5).
- *Group ID* (GID): This is the numeric ID for this group. There is normally one group defined with the group ID 0, named *root* (like the /etc/passwd record with user ID of 0). On Linux 2.2 and earlier, group IDs are maintained as 16-bit values, allowing the range 0 through to 65,535; on Linux 2.4 and later, they are stored using 32 bits.
- *User list*: This is a comma-separated list of names of users who are members of this group. (This list consists of usernames rather than user IDs, since, as noted earlier, user IDs are not necessarily unique in the password file.)

To record that the user *avr* is a member of the groups *users*, *staff*, and *teach*, we would see the following record in the password file:

```
avr:x:1001:100:Anthony Robins:/home/avr:/bin/bash
```

And the following records would appear in the group file:

```
users:x:100:
staff:x:101:mtk,avr,martinl
teach:x:104:avr,rlb,alc
```

The fourth field of the password record, containing the group ID 100, specifies membership of the group *users*. The remaining group memberships are indicated by listing *avr* once in each of the relevant records in the group file.

8.4 Retrieving User and Group Information

In this section, we look at library functions that permit us to retrieve individual records from the password, shadow password, and group files, and to scan all of the records in each of these files.

Retrieving records from the password file

The *getpwnam()* and *getpwuid()* functions retrieve records from the password file.

```
#include <pwd.h>

struct passwd *getpwnam(const char *name);
struct passwd *getpwuid(uid_t uid);
```

Both return a pointer on success, or `NULL` on error;
see main text for description of the "not found" case

Given a login name in *name*, the *getpwnam()* function returns a pointer to a structure of the following type, containing the corresponding information from the password record:

```
struct passwd {
    char *pw_name;      /* Login name (username) */
    char *pw_passwd;    /* Encrypted password */
    uid_t pw_uid;       /* User ID */
    gid_t pw_gid;       /* Group ID */
    char *pw_gecos;     /* Comment (user information) */
    char *pw_dir;       /* Initial working (home) directory */
    char *pw_shell;     /* Login shell */
};
```

The *pw_gecos* and *pw_passwd* fields of the *passwd* structure are not defined in SUSv3, but are available on all UNIX implementations. The *pw_passwd* field contains valid information only if password shadowing is not enabled. (Programmatically, the simplest way to determine whether password shadowing is enabled is to follow a successful *getpwnam()* call with a call to *getspnam()*, described shortly, to see if it returns a shadow password record for the same username.) Some other implementations provide additional, nonstandard fields in this structure.

> The *pw_gecos* field derives its name from early UNIX implementations, where this field contained information that was used for communicating with a machine running the General Electric Comprehensive Operating System (GECOS). Although this usage has long since become obsolete, the field name has survived, and the field is used for recording information about the user.

The *getpwuid()* function returns exactly the same information as *getpwnam()*, but does a lookup using the numeric user ID supplied in the argument *uid*.

Both *getpwnam()* and *getpwuid()* return a pointer to a statically allocated structure. This structure is overwritten on each call to either of these functions (or to the *getpwent()* function described below).

> Because they return a pointer to statically allocated memory, *getpwnam()* and *getpwuid()* are not reentrant. In fact, the situation is even more complex, since the returned *passwd* structure contains pointers to other information (e.g., the *pw_name* field) that is also statically allocated. (We explain reentrancy in Section 21.1.2.) Similar statements apply to the *getgrnam()* and *getgrgid()* functions (described shortly).
>
> SUSv3 specifies an equivalent set of reentrant functions—*getpwnam_r()*, *getpwuid_r()*, *getgrnam_r()*, and *getgrgid_r()*—that include as arguments both a *passwd* (or *group*) structure and a buffer area to hold the other structures to which the fields of the *passwd* (*group*) structure point. The number of bytes required for this additional buffer can be obtained using the call *sysconf(_SC_GETPW_R_SIZE_MAX)* (or *sysconf(_SC_GETGR_R_SIZE_MAX)* in the case of the group-related functions). See the manual pages for details of these functions.

According to SUSv3, if a matching *passwd* record can't be found, then *getpwnam()* and *getpwuid()* should return `NULL` and leave *errno* unchanged. This means that we should be able to distinguish the error and the "not found" cases using code such as the following:

```
struct passwd *pwd;

errno = 0;
pwd = getpwnam(name);
if (pwd == NULL) {
    if (errno == 0)
        /* Not found */;
    else
        /* Error */;
 }
```

However, a number of UNIX implementations don't conform to SUSv3 on this point. If a matching *passwd* record is not found, then these functions return `NULL` and set *errno* to a nonzero value, such as `ENOENT` or `ESRCH`. Before version 2.7, *glibc* produced the error `ENOENT` for this case, but since version 2.7, *glibc* conforms to the SUSv3 requirements. This variation across implementations arises in part because POSIX.1-1990 did not require these functions to set *errno* on error and allowed them to set *errno* for the "not found" case. The upshot of all of this is that it isn't really possible to portably distinguish the error and "not found" cases when using these functions.

Retrieving records from the group file

The *getgrnam()* and *getgrgid()* functions retrieve records from the group file.

```
#include <grp.h>

struct group *getgrnam(const char *name);
struct group *getgrgid(gid_t gid);
```

Both return a pointer on success, or `NULL` on error;
see main text for description of the "not found" case

The *getgrnam()* function looks up group information by group name, and the *getgrgid()* function performs lookups by group ID. Both functions return a pointer to a structure of the following type:

```
struct group {
    char  *gr_name;     /* Group name */
    char  *gr_passwd;   /* Encrypted password (if not password shadowing) */
    gid_t  gr_gid;      /* Group ID */
    char **gr_mem;      /* NULL-terminated array of pointers to names
                           of members listed in /etc/group */
};
```

The *gr_passwd* field of the *group* structure is not specified in SUSv3, but is available on most UNIX implementations.

As with the corresponding password functions described above, this structure is overwritten on each call to one of these functions.

If these functions can't find a matching *group* record, then they show the same variations in behavior that we described for *getpwnam()* and *getpwuid()*.

Example program

One common use of the functions that we have already described in this section is to convert symbolic user and group names into numeric IDs and vice versa. Listing 8-1 demonstrates these conversions, in the form of four functions: *userNameFromId()*, *userIdFromName()*, *groupNameFromId()*, and *groupIdFromName()*. As a convenience to the caller, *userIdFromName()* and *groupIdFromName()* also allow the *name* argument to be a (purely) numeric string; in that case, the string is converted directly to a number and returned to the caller. We employ these functions in some example programs later in this book.

Listing 8-1: Functions to convert user and group IDs to and from user and group names

users_groups/ugid_functions.c

```
#include <pwd.h>
#include <grp.h>
#include <ctype.h>
#include "ugid_functions.h"     /* Declares functions defined here */

char *          /* Return name corresponding to 'uid', or NULL on error */
userNameFromId(uid_t uid)
{
    struct passwd *pwd;

    pwd = getpwuid(uid);
    return (pwd == NULL) ? NULL : pwd->pw_name;
}

uid_t           /* Return UID corresponding to 'name', or -1 on error */
userIdFromName(const char *name)
{
    struct passwd *pwd;
    uid_t u;
    char *endptr;
    if (name == NULL || *name == '\0')  /* On NULL or empty string */
        return -1;                      /* return an error */

    u = strtol(name, &endptr, 10);      /* As a convenience to caller */
    if (*endptr == '\0')                /* allow a numeric string */
        return u;

    pwd = getpwnam(name);
    if (pwd == NULL)
        return -1;

    return pwd->pw_uid;
}

char *          /* Return name corresponding to 'gid', or NULL on error */
groupNameFromId(gid_t gid)
{
    struct group *grp;

    grp = getgrgid(gid);
    return (grp == NULL) ? NULL : grp->gr_name;
}

gid_t           /* Return GID corresponding to 'name', or -1 on error */
groupIdFromName(const char *name)
{
    struct group *grp;
    gid_t g;
    char *endptr;

    if (name == NULL || *name == '\0')  /* On NULL or empty string */
        return -1;                      /* return an error */

    g = strtol(name, &endptr, 10);      /* As a convenience to caller */
    if (*endptr == '\0')                /* allow a numeric string */
        return g;

    grp = getgrnam(name);
    if (grp == NULL)
        return -1;

    return grp->gr_gid;
}
```
users_groups/ugid_functions.c

Scanning all records in the password and group files

The *setpwent()*, *getpwent()*, and *endpwent()* functions are used to perform sequential scans of the records in the password file.

```
#include <pwd.h>

struct passwd *getpwent(void);
```

Returns pointer on success, or NULL on end of stream or error

```
void setpwent(void);
void endpwent(void);
```

The *getpwent()* function returns records from the password file one by one, returning NULL when there are no more records (or an error occurs). On the first call, *getpwent()* automatically opens the password file. When we have finished with the file, we call *endpwent()* to close it.

We can walk through the entire password file printing login names and user IDs with the following code:

```
struct passwd *pwd;

while ((pwd = getpwent()) != NULL)
    printf("%-8s %5ld\n", pwd->pw_name, (long) pwd->pw_uid);

endpwent();
```

The *endpwent()* call is necessary so that any subsequent *getpwent()* call (perhaps in some other part of our program or in some library function that we call) will reopen the password file and start from the beginning. On the other hand, if we are part-way through the file, we can use the *setpwent()* function to restart from the beginning.

The *getgrent()*, *setgrent()*, and *endgrent()* functions perform analogous tasks for the group file. We omit the prototypes for these functions because they are similar to those of the password file functions described above; see the manual pages for details.

Retrieving records from the shadow password file

The following functions are used to retrieve individual records from the shadow password file and to scan all records in that file.

```
#include <shadow.h>

struct spwd *getspnam(const char *name);
```

Returns pointer on success, or NULL on not found or error

```
struct spwd *getspent(void);
```

Returns pointer on success, or NULL on end of stream or error

```
void setspent(void);
void endspent(void);
```

We won't describe these functions in detail, since their operation is similar to the corresponding password file functions. (These functions aren't specified in SUSv3, and aren't present on all UNIX implementations.)

The *getspnam()* and *getspent()* functions return pointers to a structure of type *spwd*. This structure has the following form:

```
struct spwd {
    char *sp_namp;          /* Login name (username) */
    char *sp_pwdp;          /* Encrypted password */

    /* Remaining fields support "password aging", an optional
       feature that forces users to regularly change their
       passwords, so that even if an attacker manages to obtain
       a password, it will eventually cease to be usable. */

    long sp_lstchg;         /* Time of last password change
                               (days since 1 Jan 1970) */
    long sp_min;            /* Min. number of days between password changes */
    long sp_max;            /* Max. number of days before change required */
    long sp_warn;           /* Number of days beforehand that user is
                               warned of upcoming password expiration */
    long sp_inact;          /* Number of days after expiration that account
                               is considered inactive and locked */
    long sp_expire;         /* Date when account expires
                               (days since 1 Jan 1970) */
    unsigned long sp_flag;  /* Reserved for future use */
};
```

We demonstrate the use of *getspnam()* in Listing 8-2.

8.5 Password Encryption and User Authentication

Some applications require that users authenticate themselves. Authentication typically takes the form of a username (login name) and password. An application may maintain its own database of usernames and passwords for this purpose. Sometimes, however, it is necessary or convenient to allow users to enter their standard username and password as defined in `/etc/passwd` and `/etc/shadow`. (For the remainder of this section, we assume a system where password shadowing is enabled, and thus that the encrypted password is stored in `/etc/shadow`.) Network applications that provide some form of login to a remote system, such as *ssh* and *ftp*, are typical examples of such programs. These applications must validate a username and password in the same way that the standard *login* program does.

For security reasons, UNIX systems encrypt passwords using a *one-way encryption* algorithm, which means that there is no method of re-creating the original password from its encrypted form. Therefore, the only way of validating a candidate password is to encrypt it using the same method and see if the encrypted result matches the value stored in `/etc/shadow`. The encryption algorithm is encapsulated in the *crypt()* function.

```
#define _XOPEN_SOURCE
#include <unistd.h>

char *crypt(const char *key, const char *salt);
```

Returns pointer to statically allocated string containing encrypted password on success, or NULL on error

The *crypt()* algorithm takes a *key* (i.e., a password) of up to 8 characters, and applies a variation of the Data Encryption Standard (DES) algorithm to it. The *salt* argument is a 2-character string whose value is used to perturb (vary) the algorithm, a technique designed to make it more difficult to crack the encrypted password. The function returns a pointer to a statically allocated 13-character string that is the encrypted password.

> Details of DES can be found at *http://www.itl.nist.gov/fipspubs/fip46-2.htm*. As noted earlier, other algorithms may be used instead of DES. For example, MD5 yields a 34-character string starting with a dollar sign (`$`), which allows *crypt()* to distinguish DES-encrypted passwords from MD5-encrypted passwords.
>
> In our discussion of password encryption, we are using the word "encryption" somewhat loosely. Accurately, DES uses the given password string as an encryption key to encode a fixed bit string, while MD5 is a complex type of hashing function. The result in both cases is the same: an undecipherable and irreversible transformation of the input password.

Both the *salt* argument and the encrypted password are composed of characters selected from the 64-character set `[a-zA-Z0-9/.]`. Thus, the 2-character *salt* argument can cause the encryption algorithm to vary in any of 64 * 64 = 4096 different ways. This means that instead of preencrypting an entire dictionary and checking the encrypted password against all words in the dictionary, a cracker would need to check the password against 4096 encrypted versions of the dictionary.

The encrypted password returned by *crypt()* contains a copy of the original *salt* value as its first two characters. This means that when encrypting a candidate password, we can obtain the appropriate *salt* value from the encrypted password value already stored in `/etc/shadow`. (Programs such as *passwd(1)* generate a random *salt* value when encrypting a new password.) In fact, the *crypt()* function ignores any characters in the *salt* string beyond the first two. Therefore, we can specify the encrypted password itself as the *salt* argument.

In order to use *crypt()* on Linux, we must compile programs with the *–lcrypt* option, so that they are linked against the *crypt* library.

Example program

Listing 8-2 demonstrates how to use *crypt()* to authenticate a user. This program first reads a username and then retrieves the corresponding password record and (if it exists) shadow password record. The program prints an error message and exits if no password record is found, or if the program doesn't have permission to read from the shadow password file (this requires either superuser privilege or membership of the *shadow* group). The program then reads the user's password, using the *getpass()* function.

```
#define _BSD_SOURCE
#include <unistd.h>

char *getpass(const char *prompt);
```

Returns pointer to statically allocated input password string on success, or NULL on error

The *getpass()* function first disables echoing and all processing of terminal special characters (such as the *interrupt* character, normally *Control-C*). (We explain how to change these terminal settings in Chapter 62.) It then prints the string pointed to by *prompt*, and reads a line of input, returning the null-terminated input string with the trailing newline stripped, as its function result. (This string is statically allocated, and so will be overwritten on a subsequent call to *getpass()*.) Before returning, *getpass()* restores the terminal settings to their original states.

Having read a password with *getpass()*, the program in Listing 8-2 then validates that password by using *crypt()* to encrypt it and checking that the resulting string matches the encrypted password recorded in the shadow password file. If the password matches, then the ID of the user is displayed, as in the following example:

```
$ su                          Need privilege to read shadow password file
Password:
# ./check_password
Username: mtk
Password:                     We type in password, which is not echoed
Successfully authenticated: UID=1000
```

> The program in Listing 8-2 sizes the character array holding a username using the value returned by *sysconf(_SC_LOGIN_NAME_MAX)*, which yields the maximum size of a username on the host system. We explain the use of *sysconf()* in Section 11.2.

Listing 8-2: Authenticating a user against the shadow password file

users_groups/check_password.c

```
#define _BSD_SOURCE     /* Get getpass() declaration from <unistd.h> */
#define _XOPEN_SOURCE   /* Get crypt() declaration from <unistd.h> */
#include <unistd.h>
#include <limits.h>
#include <pwd.h>
#include <shadow.h>
#include "tlpi_hdr.h"

int
main(int argc, char *argv[])
{
    char *username, *password, *encrypted, *p;
    struct passwd *pwd;
    struct spwd *spwd;
    Boolean authOk;
    size_t len;
    long lnmax;
    lnmax = sysconf(_SC_LOGIN_NAME_MAX);
    if (lnmax == -1)                    /* If limit is indeterminate */
        lnmax = 256;                    /* make a guess */

    username = malloc(lnmax);
    if (username == NULL)
        errExit("malloc");

    printf("Username: ");
    fflush(stdout);
    if (fgets(username, lnmax, stdin) == NULL)
        exit(EXIT_FAILURE);             /* Exit on EOF */

    len = strlen(username);
    if (username[len - 1] == '\n')
        username[len - 1] = '\0';       /* Remove trailing '\n' */

    pwd = getpwnam(username);
    if (pwd == NULL)
        fatal("couldn't get password record");
    spwd = getspnam(username);
    if (spwd == NULL && errno == EACCES)
        fatal("no permission to read shadow password file");

    if (spwd != NULL)           /* If there is a shadow password record */
        pwd->pw_passwd = spwd->sp_pwdp;     /* Use the shadow password */

    password = getpass("Password: ");

    /* Encrypt password and erase cleartext version immediately */

    encrypted = crypt(password, pwd->pw_passwd);
    for (p = password; *p != '\0'; )
        *p++ = '\0';

    if (encrypted == NULL)
        errExit("crypt");

    authOk = strcmp(encrypted, pwd->pw_passwd) == 0;
    if (!authOk) {
        printf("Incorrect password\n");
        exit(EXIT_FAILURE);
    }

    printf("Successfully authenticated: UID=%ld\n", (long) pwd->pw_uid);

    /* Now do authenticated work... */

    exit(EXIT_SUCCESS);
}
```

users_groups/check_password.c

Listing 8-2 illustrates an important security point. Programs that read a password should immediately encrypt that password and erase the unencrypted version from memory. This minimizes the possibility of a program crash producing a core dump file that could be read to discover the password.

> There are other possible ways in which the unencrypted password could be exposed. For example, the password could be read from the swap file by a privileged program if the virtual memory page containing the password is swapped out. Alternatively, a process with sufficient privilege could read `/dev/mem` (a virtual device that presents the physical memory of a computer as a sequential stream of bytes) in an attempt to discover the password.
>
> The *getpass()* function appeared in SUSv2, which marked it LEGACY, noting that the name was misleading and the function provided functionality that was in any case easy to implement. The specification of *getpass()* was removed in SUSv3. It nevertheless appears on most UNIX implementations.

8.6 Summary

Each user has a unique login name and an associated numeric user ID. Users can belong to one or more groups, each of which also has a unique name and an associated numeric identifier. The primary purpose of these identifiers is to establish ownership of various system resources (e.g., files) and permissions for accessing them.

A user's name and ID are defined in the `/etc/passwd` file, which also contains other information about the user. A user's group memberships are defined by fields in the `/etc/passwd` and `/etc/group` files. A further file, `/etc/shadow`, which can be read only by privileged processes, is used to separate the sensitive password information from the publicly available user information in `/etc/passwd`. Various library functions are provided for retrieving information from each of these files.

The *crypt()* function encrypts a password in the same manner as the standard *login* program, which is useful for programs that need to authenticate users.

8.7 Exercises

8-1. When we execute the following code, we find that it displays the same number twice, even though the two users have different IDs in the password file. Why is this?

```
printf("%ld %ld\n", (long) (getpwnam("avr")->pw_uid),
                    (long) (getpwnam("tsr")->pw_uid));
```

8-2. Implement *getpwnam()* using *setpwent()*, *getpwent()*, and *endpwent()*.