Advanced Programming in the UNIX Environment — *System Datas Files and Information*

Hop Lee
hoplee@bupt.edu.cn

Contents

1 Password File

- The UNIX password file `/etc/passwd` contains several fields. These fields are contained in a `passwd` structure defined in `pwd.h`:

```
structure passwd
{
  char  *pw_name;
  char  *pw_passwd;
  uid_t  pw_uid;
  gid_t  pw_gid;
  time_t pw_change;
  time_t pw_expire;
  char  *pw_class;
  char  *pw_gecos;
  char  *pw_dir;
  char  *pw_shell;
};
```

- The field delimiter in `/etc/passwd` is a colon.
- The sub-delimiter in field `RealName` is a comma. The sub-fields include: the user's full name, office location, office phone number, and home phone number.
- There are two POSIX.1 functions can fetch entries from the password file:

```c
#include <sys/types.h>
#include <pwd.h>
struct passwd *getpwuid(uid_t uid);
struct passwd *getpwnam(const char *name);
```

- Both functions return a pointer to a `passwd` structure that the function will fill in.
- When we need to go through the entire password file, the following functions can be used for this:

```c
#include <sys/types.h>
#include <pwd.h>
struct passwd *getpwent(void);
void setpwent(void);
void endpwent(void);
```

- `getpwent` returns a pointer to the next password file entry that it has filled in.
- The function `setpwent` rewinds the file it uses, and `endpwent` closes these files.
- Example (Figure 6.2, `datafiles/getpwnam.c`).

2 Shadow Passwords

- Because there are some brute force approach to hack the password file, some system store the encrypted password in another file—**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).
- 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.

```c
#include <shadow.h>
struct spwd *getspnam(const char *name);
struct spwd *getspent(void);
void setspent(void);
void endspent(void);
```

- On FreeBSD 8.0 and Mac OS X 10.6.8, there is no shadow password structure.

3 Group File

- The UNIX group file contains 4 fields separated by colon. These fields are contained in a `group` structure defined in `grp.h`.

```c
structure group
{
  char *gr_name;
  char *gr_passwd;
  gid_t gr_gid;
  char **gr_mem;
};
```

- The field `gr_mem` is an array of pointers to the username that belong to this group, terminated by a null pointer.
- There are two POSIX.1 functions can fetch entries from the group file:

```
#include <grp.h>
struct group *getgrgid(gid_t gid);
struct group *getgrnam(const char *name);
```

- If we want to search the entire group file we need some additional functions:

```
#include <grp.h>
struct group *getgrent(void);
void setgrent(void);
void endgrent(void);
```

- Another three functions are provided to fetch and set the supplementary group IDs:

```
#include <unistd.h>
int getgroups(int gidsetsize, gid_t grplist[]);
int setgroups(int ngroups, const gid_t grplist[]);
int initgroups(const char *username, gid_t basegid);
```

- The function `getgroups` fills up to *gidsetsize* elements in the array *grplist* with the supplementary group IDs, and return the number of supplementary group IDs stored.
- `setgroups` can be called by the root to set the supplementary group ID list for the calling process.
- `initgroups` reads the entire `group` file and determines the group membership for username. Then calls `setgroups` to initialize the supplementary group ID list for the user.

4 Other Data Files

- Numerous other files are used by UNIX systems in normal day-to-day operation.
- 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.
  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. 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.

5 Login Accounting

- Two data file that have been provided with most UNIX systems are `utmp` file, which keeps track of all the users currently logged in, and the `wtmp` file, which keeps track of all logins and logouts.
- The corresponding data structure is:

```
struct utmp {
  char ut_line[8];
  char ut_name[8];
  long ut_time;
};
```

6 System Identification

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

```
#include <sys/utsname.h>
int uname(struct utsname *name);
```

- Return nonnegative if OK, -1 on error.
- The function will fill the structure.
- The corresponding data structure is:

```
struct utsname{
  char sysname[9];
  char nodename[9];
  char release[9];
  char version[9];
  char machine[9];
  char domainname[9];
};
```

- Berkeley-derived systems provide the `gethostname` function to return just the name of the host:

```
#include <unistd.h>
int gethostname(char *name, int namelen);
```

- There is also a similar `sethostname` function used to set host name only by root.

7 Time and Date Routines

- The basic time service provide by the UNIX kernel is the number of seconds that have passed since **Epoch**. We call it calendar times.
- The `time` function returns the current time and date:

```
#include <time.h>
time_t time(time_t *calptr);
```

- The real-time extensions to POSIX.1 added support for multiple system clocks. In SUSv4, 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 the table shown bellow.

Table 1: Clock type identifiers

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

- 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.

```
1 #include <sys/time.h>
2 int clock_gettime(clockid_t clock_id,
3                   struct timespec *tsp);
```

- When the clock ID is set to CLOCK_REALTIME, the clock_gettime function provides similar functionality to the time function except 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.

```
1 #include <sys/time.h>
2 int clock_getres(clockid_t clock_id,
3                  struct timespec *tsp);
```

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

```
1 #include <sys/time.h>
2 int clock_settime(clockid_t clock_id,
3                   const struct timespec *tsp);
```

- We need the appropriate privileges to change a clock's time. Some clocks, however, can't be modified.
- SUSv4 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.

```
1 #include <sys/time.h>
2 int gettimeofday(struct timeval *restrict tp,
3                  void *restrict tzp);
```

- This 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:

```
1 struct timeval {
2   time_t tv_sec;   /* seconds */
3   long   tv_usec;  /* microseconds */
4 };
```

- The only legal value for *tzp* is `NULL`; any other value results in unspecified behavior.
- There are several time functions used to convert the large integer value to a human readable time and date.
- The functions `localtime` and `gmtime` convert a calendar time into what's called a **broken-down time**, a `tm` structure shown at next page.

```
#include <time.h>
struct tm *gmtime(const time_t *calptr);
struct tm *localtime(const time_t *calptr);
struct tm{          /* a broken-down time */
  int tm_sec;       /* [0, 60] */
  int tm_min;       /* [0, 59] */
  int tm_hour;      /* [0, 23] */
  int tm_mday;      /* [1, 31] */
  int tm_mon;       /* [0, 11] */
  int tm_year;      /* year-1900 */
  int tm_wday;      /* [0, 6] */
  int tm_yday;      /* [0, 365] */
  int tm_isdst;     /* <0, 0, >0 */
};
```

- The `localtime` converts the calendar time to the local time (taking into account the local time zone and daylight saving time flag), and the `gmtime` converts the calendar time into UTC.
- The `mktime` function do the reverse thing:

```
#include <time.h>
time_t mktime(struct tm *tmptr);
```

- The `asctime` and `ctime` functions produce the familiar 28-byte (42 bytes in SC) string that is similar to the default output of the `date`(1) command:

```
#include <time.h>
char *asctime(const struct tm *tmptr);
char *ctime(const time_t *calptr);
```

- The most sophisticate function is `strftime`. It is a `printf`-like function for time values:

```
#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);
```

- The final parameter *tmptr*is the time value to be formatted. The formatted result is store in the array *buf* whose size is *maxsize* characters. The *fmt* parameter controls the formatting of the time value, like the `printf` function. Figure 6.10 describes the 37 ISO C conversion specifiers.

- Example (Figure 6.11, `datafiles/strftime.c`).

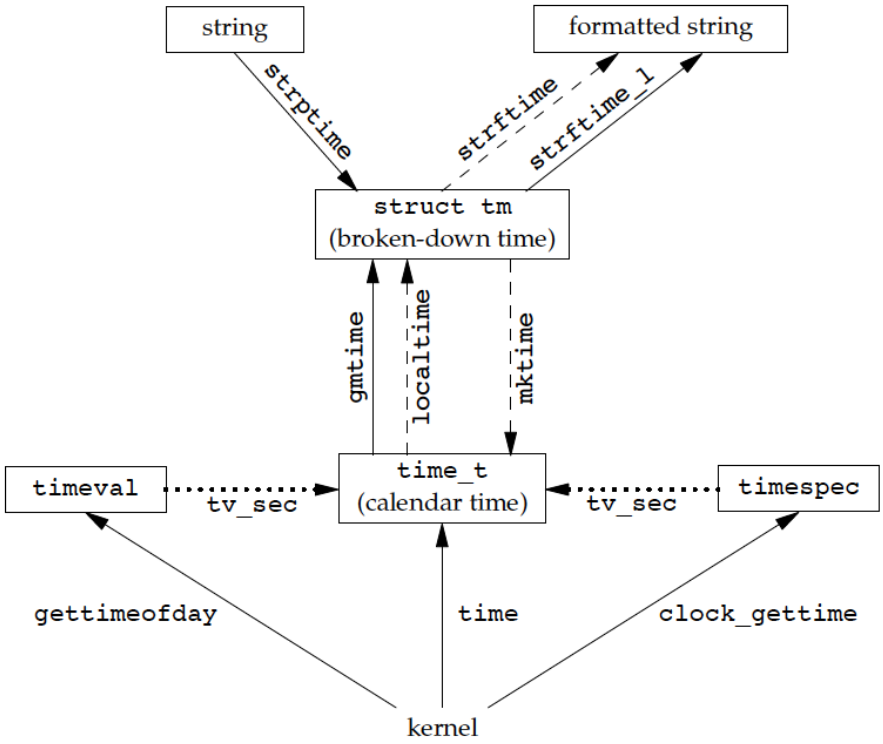

Figure 1: Relationship of the various time functions

- 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);
```

The End of Chapter 6.