Advanced Programming in the UNIX Environment — *Files and Directories*

Hop Lee
hoplee@bupt.edu.cn

School of Information and Communication Engineering

Table of Contents I

Table of Contents II

File Status I

- The stat function returns a structure of information about the given file.

```
#include <sys/types.h>
#include <sys/stat.h>
int stat(const char *restrict pathname,
         struct stat *restrict buf);
int fstat(int filedes, struct stat *buf);
int lstat(const char *restrict pathname,
          struct stat *restrict buf);
int fstatat(int fd, const char *restrict pathnam
            struct stat *restrict buf, int flag)
```

File Status II

- Given a *pathname*, the `stat` function returns a structure of information about the named file. The `fstat` function obtains information about the file that is already open on the descriptor *fd*. The `lstat` function is similar to `stat`, but when the named file is a symbolic link, `lstat` returns information about the symbolic link itself, not the file referenced by the symbolic link.
- The `fstatat` function provides a way to return the file statistics for a pathname relative to an open directory represented by the *fd* argument. The *flag* argument controls whether symbolic links are followed.

File Status III

- The *buf* argument is a pointer to a structure that we must supply. The functions fill in the structure. The definition of the structure can differ among implementations, but it could look like

```
struct stat {
  mode_t st_mode; /* file type & mode (permissions) */
  ino_t st_ino; /* i-node number (serial number) */
  dev_t st_dev; /* device number (file system) */
  dev_t st_rdev; /* device number for special files */
  nlink_t st_nlink; /* number of links */
  uid_t st_uid; /* user ID of owner */
  gid_t st_gid; /* group ID of owner */
  off_t st_size; /* size in bytes, for regular files */
  struct timespec st_atim; /* time of LAT */
  struct timespec st_mtim; /* time of LMT */
  struct timespec st_ctim; /* time of LCT */
  blksize_t st_blksize; /* best I/O block size */
```

File Status IV

```
    blkcnt_t st_blocks; /* no. of disk blocks allocated */
};
```

- The timespec structure type defines time in terms of seconds and nanoseconds. It includes at least the following fields:

```
time_t tv_sec;
long tv_nsec;
```

- We'll go through each member of this structure to examine the attributes of a file in the rest of this chapter.

File Types I

- There are seven file types in most UNIX system:
  - Regular file
  - Directory file
  - Character device file
  - Block device file
  - FIFO
  - Socket
  - Symbolic link.
- Example (Figure 4.3, `filedir/filetype.c`).
- The type of a file is encoded in the `st_mode` member of the `stat` structure. We can determine the file type by those macros listed in Figure 4.1.

File Types II

- Next table shows the counts and percentages for a Linux system that is used as a single-user workstation. This data was obtained from the program shown in Section 4.22.

Table: Counts and percentages of different file types

File type	Count	Percentage(%)
regular file	415,803	79.77
directory	62,197	11.93
symbolic link	40,018	8.25
character special	155	0.03
block special	47	0.01
socket	45	0.01
FIFO	0	0.00

Set-User-ID and Set-Group-ID

- Each process has six or more IDs associated with it: real, effective, and saved set.
- The real ID identify who we really are.
- The effective ID and supplementary group ID determine our file access permissions.
- The saved set ID contain copies of the effective ID when a program is executed. See §8.11.
- The **set-user-ID** bit and **set-group-ID** bit are contained in the file's `st_mode` value. These two bits can be tested against the constants `S_ISUID` and `S_ISGID`.

File Access Permissions I

- The `st_mode` value also encodes the access permission bits for the file.
- Read permission and execute permission for a directory mean different things. Read permission let us can use "`ls`" command, execute permission lets us pass through the directory when it is a component of a pathname that we trying to access.
- The execute permission bit for a directory is often called the search bit.
- We must have write permission for a file to specify the `O_TRUNC` flag in the `open` function.

File Access Permissions II

- To delete an existing file, we need write and execute permission in the directory containing the file, but do NOT need read or write permission for that file.
- The tests perform by the kernel for file access privilege is a short-cut test.

Ownership of New Files and Directories

- The user ID of a new file is set to the effective user ID of the process.
- `POSIX.1` allows an implementation to choose one of the following options to determine the group ID of a new file:
  - the effective group ID of the process
  - the group ID of the directory in which the new file is being created.

access and faccessat Functions I

- When a process want to test accessibility based on the real user ID and real group ID, access function may be useful.

```
#include <unistd.h>
int access (const char *pathname, int mode);
int faccessat(int fd, const char *pathname,
              int mode, int flag);
```

- The *mode* is the bitwise OR of any of the constant: R_OK, W_OK, X_OK, F_OK.

access and faccessat Functions II

- The faccessat function behaves like access when the *pathname* argument is absolute or when the *fd* has the value AT_FDCWD and the *pathname* is relative. Otherwise, faccessat evaluates the *pathname* relative to the open directory referenced by the *fd*.
- The *flag* argument can be used to change the behavior of faccessat. If the AT_EACCESS flag is set, the access checks are made using the effective user and group IDs of the calling process instead of the real user and group IDs
- Example (Figure 4.8, filedir/access.c).

umask Function

- The umask function sets the file mode creation mask for the process and returns the previous value.

```
#include <sys/stat.h>
mode_t umask(mode_t cmask);
```

- The *cmask* parameter is formed as the bitwise OR of any of the 9 constants from Figure 4.6.
- Any bits that are on in the file mode creation mask are turned off in the file's mode.
- Example (Figure 4.9, filedir/umask.c).

chmod, fchmod, and fchmodat Functions I

- Following functions allow us to change the file access permission for an existing file.

```
#include <sys/stat.h>
int chmod(const char *pathname, mode_t mode);
int fchmod(int fd, mode_t mode);
int fchmodat(int fd, const char *pathname,
                 mode_t mode, int flag);
```

- The *flag* argument can be used to change the behavior of `fchmodat` —when the `AT_SYMLINK_NOFOLLOW` flag is set, `fchmodat` doesn't follow symbolic links.

chmod, fchmod, and fchmodat Functions II

- To change the permission bits of a file, the effective user ID of the process must equal the owner of the file, or the process must have superuser permission.
- The *mode* is specified as the bitwise OR of the constants shown in Figure 4.11.
- Example (Figure 4.12, `filedir/changemod.c`).

Sticky Bit

- If the sticky bit of a program was set, then the first time the program was executed a copy of the program's text was saved in the swap area when the process terminated. This caused the program to load into memory faster the next time it was executed.
- If the sticky bit was set for a directory, a file in the directory can be removed or renamed only if the user has write permission for the directory, and either:
  - owns the file;
  - owns the directory, or
  - is the superuser.
- The sticky bit is NOT defined by `POSIX.1`.

chown, fchown, fchownat, and lchown Functions I

- These functions allow us to change the use ID and the group ID of a file.

```
#include <sys/types.h>
#include <unistd.h>
int chown (const char *pathname, uid_t owner, gid_t group);
int fchown (int fd, uid_t owner, gid_t group);
int fchownat(int fd, const char *pathname,uid_t owner,
             gid_t group, int flag);
int lchown (const char *pathname, uid_t owner, gid_t group);
```

- POSIX.1 allows either form of operation, depending on the value of _POSIX_CHOWN_RESTRICTED:
  - Only the superuser can change the ownership of a file, or
  - Any user can change the ownership of any files they own.

chown, fchown, fchownat, and lchown Functions II

- Historically, BSD-based systems have enforced the restriction that only the superuser can change the ownership of a file. This is to prevent users from giving away their files to others, thereby defeating any disk space quota restrictions.
- If these functions are called by a process other than a superuser process, on successful return, both the set-user-ID and the set-group-ID bits are cleared.

File Size

- The st_size member of the stat structure contains the size of the file in bytes. This field is meaningful only for regular files, directories, and symbolic links.
- st_blksize and st_blocks means the preferred block size for I/O for the file and the actual number of 512-byte blocks that are allocated.
- Examples on textbook.

File Truncation

- Following functions truncate an existing file to *length* bytes, the data beyond *length* is no longer accessible.

```
#include <sys/types.h>
#include <unistd.h>
int truncate (const char *pathname, off_t length);
int ftrunctate (int fd, off_t length);
```

Filesystems I

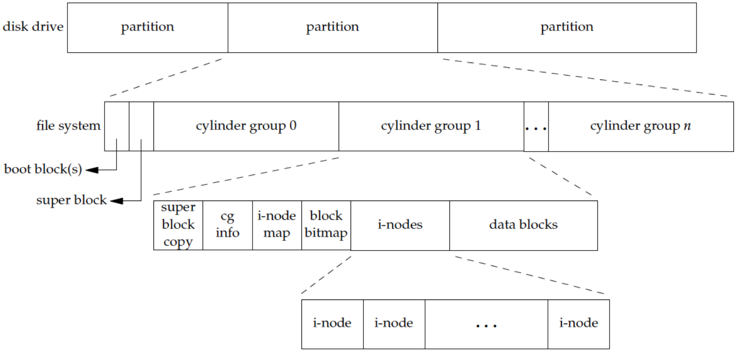

Figure: Disk drive, partitions, and a file system

Filesystems II

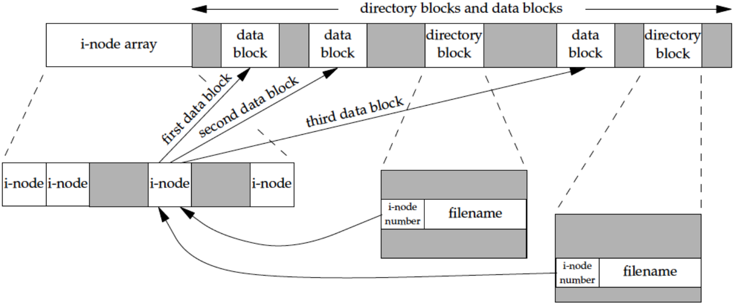

Figure: Cylinder group's i-nodes and data blocks in more detail

Filesystems III

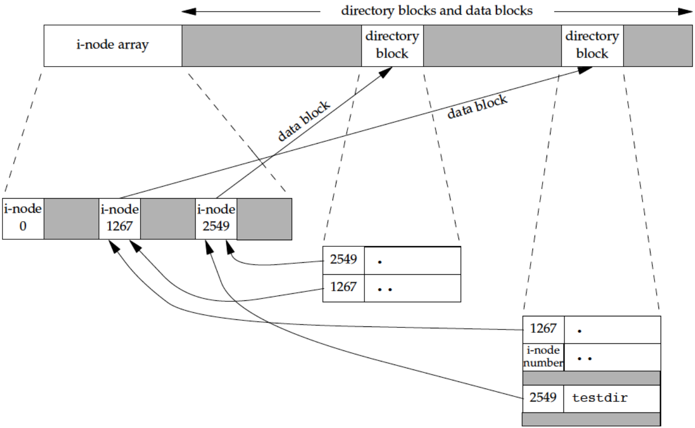

Figure: Sample cylinder group after creating the directory testdir

Filesystems IV

- Every i-node has a link count that contains the number of directory entries that point to the i-node. Only when the link count decrease to 0 can the file be deleted.
- The `st_nlink` member of `stat` structure contains the link count. The files share the same i-node called **hard link**.
- The **symbolic link** file has its own i-node. And the actual content of a symbolic file is the pathname of the file that it points to.
- The i-node contains all the information about the file: the file type, the file's access permission bits, the size of the file, pointers to the data blocks for the file, and so on. The data type for the i-node number is `ino_t`.

Filesystems V

- A directory entry contains only two items: filename and the i-node number.
- The actual operation of rename a file is unlink old directory entry and create a new directory entry points to the existing i-node.

link, linkat, unlink, unlinkat, and remove Functions I

- The link and the linkat function can create a link to an existing file.

```
#include <unistd.h>
int link(const char *src, const char *dst);
int linkat(int sfd, const char *src,
           int dfd, const char *dst,
           int flag);
```

- The creation of the new directory entry and the increment of the link count must be an atomic operation.

link, linkat, unlink, unlinkat, and remove Functions II

- To remove an existing directory entry we call the unlink function:

```
#include <unistd.h>
int unlink(const char *pathname);
int unlinkat(int fd, const char *pathname,
             int flag);
```

- If the link count is NOT 0 after unlink call, the actual data of the file will NOT be deleted.
- As long as some other process has this file open, its contents will not be deleted.
- Example (Figure 4.16, filedir/unlink.c).

link, linkat, unlink, unlinkat, and remove Functions III

- We can also unlink a file or directory with the remove function. For a file, it is identical to unlink, for a directory it is identical to rmdir.

```
#include <stdio.h>
int remove(const char *pathname);
```

rename and renameat Functions I

- A file or a directory is renamed with either the rename or renameat function.

```
#include <stdio.h>
int rename(const char *oldname, const char *newname);
int renameat(int oldfd, const char *oldname, int newfd,
             const char *newname);
```

- Both return: 0 if OK, −1 on error.
- There are several conditions to describe for these functions, depending on whether *oldname* refers to a file, a directory, or a symbolic link. We must also describe what happens if *newname* already exists.

`rename` and `renameat` Functions II

1. If *oldname* specifies a file that is not a directory, then we are renaming a file or a symbolic link. In this case, if *newname* exists, it cannot refer to a directory. Then it been removed, and *oldname* is renamed to *newname*. We must have write permission for the directories containing *oldname* and *newname*.
2. If *oldname* specifies a directory, then we are renaming a directory. If *newname* exists, it must refer to a directory, and that directory must be empty. Then it been removed, and *oldname* is renamed to *newname*. Additionally, when we're renaming a directory, *newname* cannot contain a path prefix that names *oldname*.
3. If either *oldname* or *newname* refers to a symbolic link, then the link itself is processed, not the file to which it points.

rename and renameat Functions III

4. We can't rename dot or dot-dot. More precisely, neither dot nor dot-dot can appear as the last component of *oldname* or *newname*.
5. As a special case, if *oldname* and *newname* refer to the same file, the function returns successfully without changing anything.

- If *newname* already exists, we need permissions as if we were deleting it. Also, because we're removing the directory entry for *oldname* and possibly creating a directory entry for *newname*, we need write permission and execute permission in the directories containing *oldname* and *newname*.

rename and renameat Functions IV

- The renameat function provides the same functionality as the rename function, except when either *oldname* or *newname* refers to a relative pathname. In this case, it is evaluated relative to the directory referenced by *oldfd*. Similarly, *newname* is evaluated relative to the directory referenced by *newfd* if *newname* specifies a relative pathname. Either the *oldfd* or *newfd* arguments (or both) can be set to AT_FDCWD to evaluate the corresponding pathname relative to the current working directory.

Symbolic Link

- **Symbolic links** were introduced to get around the limitations of hard links:
  - Hard links normally require that the link and the file reside in the same filesystem;
  - Only the superuser can create a hard link to a directory.
- Symbolic link are typically used to move a file or an entire directory hierarchy to some other location on a system.
- Figure 4.17 summarizes whether the function described in this chapter follow a symbolic link or not.
- Whether `chown` follows a symbolic link or not depends on the implementation.
- Example: a loop of symbolic link (Figure 4.18).

Creating and Reading Symbolic Links

- A symbolic link is created with the symlink or symlinkat function.

```
#include <unistd.h>
int symlink(const char *src, const char *dst);
int symlinkat(const char *src, int fd, const char *dst);
```

- Since the open function follows a symbolic link, we need a way to open the link itself and read the name in the link. The readlink and readlinkat functions do this.

```
#include <unistd.h>
ssize_t readlink(const char* restrict pathname,
                 char* restrict buf, size_t bufsize);
ssize_t readlinkat(int fd, const char * restrict pathname,
                   char* restrict buf, size_t bufsize);
```

- This functions combines the actions of open, read, and close.

File Time

- Three time fields are maintained for each file:

  `ST_ATIME`: last access time of file data;

  `ST_MTIME`: last modification time of file data;

  `ST_CTIME`: last change time of i-node status.

- The system does not maintain the last access time for an `i-node`.
- The `ls`(1) command displays or sorts only on one of the three time value. By default **LMT**, the `-u` option means **LAT**, and the `-c` means **LCT**.
- Figure 4.20 summarizes the effects of the various functions that we've described on these three times.

futimens, utimesat, and utime Functions I

- The LAT and the LMT of a file will be set to the greatest value supported by the filesystem that is not greater than the specified time with these functions in nanosecond precision:

```
#include <sys/stat.h>
int futimens(int fd, const struct timespec times[2]);
int utimensat(int dirfd, const char *path,
              const struct timespec times[2],
              int flags);

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

- In both functions, the 1st element of the *times* array contains the LAT, and the 2nd element contains the LMT. The two time values are calendar times.

futimens, utimesat, and utime Functions II

- Timestamps can be specified in one of four ways:
  1. The *times* argument is a null pointer. In this case, both timestamps are set to the current time.
  2. If either `tv_nsec` field has the special value `UTIME_NOW`, the corresponding timestamp is set to the current time. The corresponding `tv_sec` field is ignored.
  3. If either `tv_nsec` field has the special value `UTIME_OMIT`, then the corresponding timestamp is left unchanged. The corresponding `tv_sec` field is ignored.
  4. The *times* argument points to an array of two timespec structures and the `tv_nsec` field contains a value other than `UTIME_NOW` or `UTIME_OMIT`. Then the corresponding timestamp is set to the value specified by the corresponding `tv_sec` and `tv_nsec` fields.

futimens, utimesat, and utime Functions III

- The privileges required to execute these functions depend on the value of the *times* argument.
- Both futimens and utimensat are included in POSIX.1. A third function, utimes, is included in the Single UNIX Specification as part of the XSI option.

```
#include <sys/time.h>
int utimes (const char *pathname,
            const struct timeval times[2]);

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

futimens, utimesat, and utime Functions IV

- The utimes function operates on a pathname. The *times* argument is expressed in seconds and microseconds.
- Note that we are unable to specify a value for the changed-status time as this field is automatically updated when the utime functions family are called.
- Example (Figure 4.21, filedir/zap.c).

mkdir, mkdirat, and rmdir Functions

- Directories are created and deleted with mkdir and rmdir functions.

```
#include <sys/stat.h>
int mkdir(const char *pathname, mode_t mode);
int mkdirat(int fd, const char *pathname,
            mode_t mode);
```

- Be careful of *mode* parameter.

```
#include <unistd.h>
int rmdir (const char *pathname);
```

- The *pathname* parameter must be an empty directory.
- When this function is called, the *pathname* will be locked.

Reading Directories I

- Directories can be read by anyone who has access permission to read the directory. But only kernel can write to a directory.
- A set of directory routines were developed and are part of POSIX.1.

```
struct dirent{
  ino_t  d_ino;
  char   d_name[256];
};

#include <dirent.h>
DIR *opendir(const char *pathname);
DIR *fdopendir(int fd);
struct dirent *readdir(DIR *dp);
```

Reading Directories II

```
void rewinddir(DIR *dp);
int closedir(DIR *dp);
long telldir(DIR *dp);
void seekdir(DIR *dp, long loc);
```

- The `telldir` and `seekdir` functions are not part of the base POSIX.1 standard. They are included in the XSI option in the SUS, so all conforming UNIX System implementations are expected to provide them.
- The ordering of entries within the directory is implementation dependent and is usually not alphabetical.
- Example (Figure 4.22, `filedir/ftw8.c`).

chdir, fchdir, and getcwd Functions I

- Every process has a current working directory.
- We can change the current working directory of the calling process by calling the chdir or fchdir functions.

```
#include <unistd.h>
int chdir (const char *pathname);
int fchdir (int fd);
```

- Both return 0 if OK, -1 on error.
- Since the current working directory is an attribute of a process it cannot affect others processes.
- Example (Figure 4.23, filedir/mycd.c).

chdir, fchdir, and getcwd Functions II

- Unfortunately, all the kernel maintains for each process is the i-node number and device id for the CWD. So we need a function that can obtain the entire absolute pathname of the CWD by the giving i-node number.

```
#include <unistd.h>
char *getcwd (char *buf, size_t size);
```

- Example (Figure 4.24, filedir/cdpwd.c).

Device Special Files I

- Every filesystem is known by its major and minor device number. This device number is encoded in the primitive system data type `dev_t`.
- The major number identifies the device driver and sometimes encodes which peripheral board to communicate with; the minor number identifies the specific subdevice.
- We can usually access the major and minor device numbers through two macros: `major` and `minor`.
- Linux store the device number in a 16-bit integer with higher 8 bits for the major and lower 8 bits for minor device number.
- These two macros are defined in `sys/sysmacros.h`.

Device Special Files II

- The st_dev value for every filename on a system is the device number of the filesystem containing that filename and its i-node.
- Only character and block special device files have an st_rdev value. This value contains the device number for the actual device.
- Example (Figure 4.25, filedir/devrdev.c).

Summary

- This chapter centered around the `stat` function. We've gone through each member in the `stat` structure in detail.
- A through understanding of all the properties of files and directories and all the functions that operate on them is essential to UNIX programming.

The End

The End of Chapter 4.