

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

Hop Lee
hoplee@bupt.edu.cn

Contents

1	File Status	2
2	File Types	3
3	Set-User-ID and Set-Group-ID	3
4	File Access Permissions	3
5	Ownership of New Files and Directories	4
6	access and faccessat Functions	4
7	umask Function	4
8	chmod, fchmod, and fchmodat Functions	5
9	Sticky Bit	5
10	chown, fchown, fchownat, and lchown Functions	5
11	File Size	6
12	File Truncation	6
13	Filesystems	6
14	link, linkat, unlink, unlinkat, and remove Functions	8
15	rename and renameat Functions	9
16	Symbolic Link	9
17	Creating and Reading Symbolic Links	10
18	File Time	10
19	futimens, utimesat, and utimes Functions	10
20	mkdir, mkdirat, and rmdir Functions	11
21	Reading Directories	12

1 File Status

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

```

1 #include <sys/types.h>
2 #include <sys/stat.h>
3 int stat(const char *restrict pathname,
4          struct stat *restrict buf);
5 int fstat(int filedes, struct stat *buf);
6 int lstat(const char *restrict pathname,
7           struct stat *restrict buf);
8 int fstatat(int fd, const char *restrict pathname,
9             struct stat *restrict buf, int flag);

```

- 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.
- 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

```

1 struct stat {
2     mode_t st_mode; /* file type & mode (permissions) */
3     ino_t st_ino; /* i-node number (serial number) */
4     dev_t st_dev; /* device number (file system) */
5     dev_t st_rdev; /* device number for special files */
6     nlink_t st_nlink; /* number of links */
7     uid_t st_uid; /* user ID of owner */
8     gid_t st_gid; /* group ID of owner */
9     off_t st_size; /* size in bytes, for regular files */
10    struct timespec st_atim; /* time of LAT */
11    struct timespec st_mtim; /* time of LMT */
12    struct timespec st_ctim; /* time of LCT */
13    blksize_t st_blksize; /* best I/O block size */
14    blkcnt_t st_blocks; /* no. of disk blocks allocated */
15 };

```

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

```

1 time_t tv_sec;
2 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.

2 File Types

- 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.
- 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 1: 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

3 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`.

4 File Access Permissions

- 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.
- 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.

5 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.

6 access and faccessat Functions

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

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

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

7 umask Function

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

```
1 #include <sys/stat.h>
2 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`).

8 chmod, fchmod, and fchmodat Functions

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

```
1 #include <sys/stat.h>
2 int chmod(const char *pathname, mode_t mode);
3 int fchmod(int fd, mode_t mode);
4 int fchmodat(int fd, const char *pathname,
5             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.
- 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`).

9 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.

10 chown, fchown, fchownat, and lchown Functions

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

```
1 #include <sys/types.h>
2 #include <unistd.h>
3 int chown(const char *pathname, uid_t owner, gid_t group);
4 int fchown(int fd, uid_t owner, gid_t group);
5 int fchownat(int fd, const char *pathname, uid_t owner,
6             gid_t group, int flag);
7 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.

- 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.

11 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.

12 File Truncation

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

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

13 Filesystems

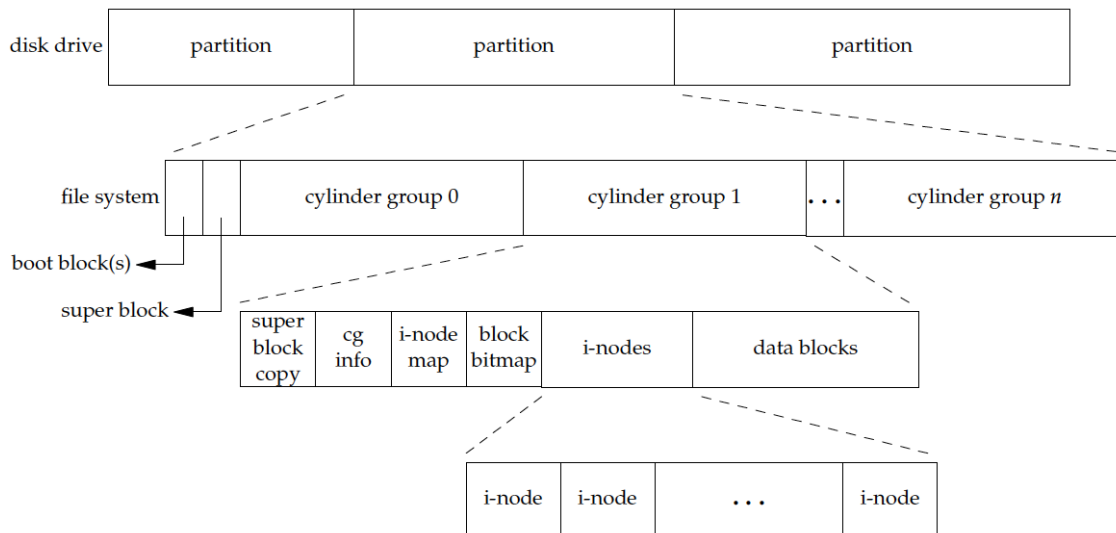


Figure 1: Disk drive, partitions, and a file system

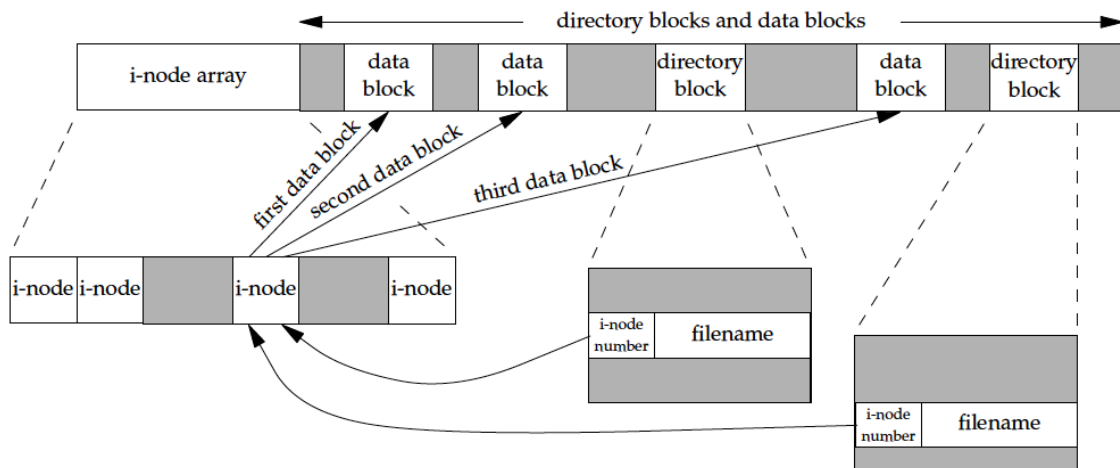


Figure 2: Cylinder group's i-nodes and data blocks in more detail

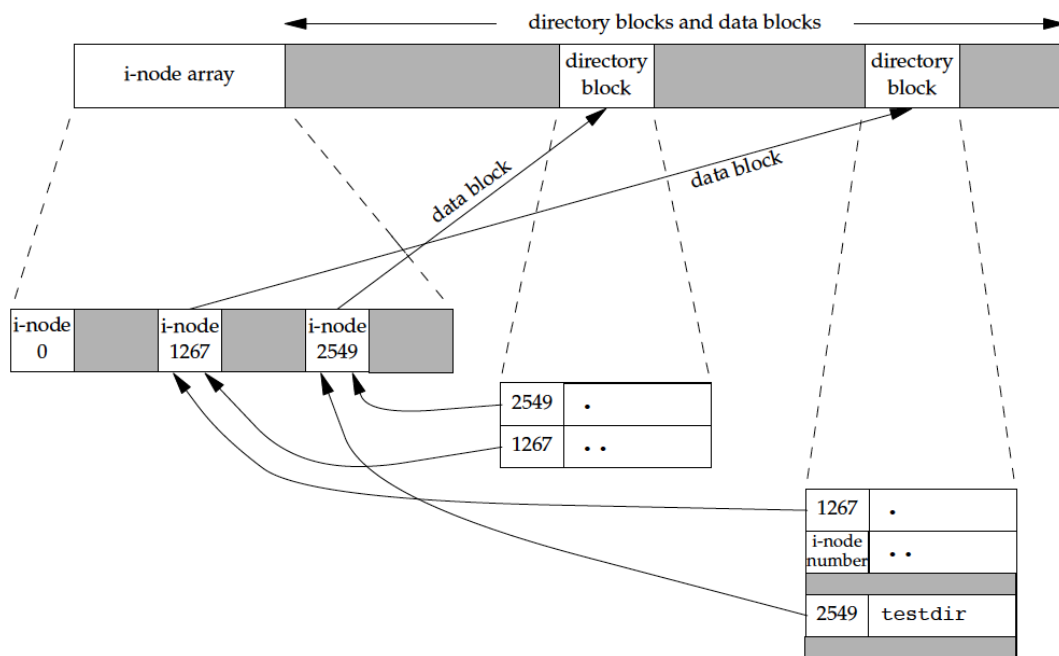


Figure 3: Sample cylinder group after creating the directory `testdir`

- 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`.
- 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.

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

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

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

- The creation of the new directory entry and the increment of the link count must be an *atomic* operation.
- To remove an existing directory entry we call the `unlink` function:

```
1 #include <unistd.h>
2 int unlink(const char *pathname);
3 int unlinkat(int fd, const char *pathname,
4           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`).
- 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`.

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


15 rename and renameat Functions

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

```
1 #include <stdio.h>
2 int rename(const char *oldname, const char *newname);
3 int renameat(int oldfd, const char *oldname, int newfd,
4             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.
 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.
 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*.
- 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.

16 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).

17 Creating and Reading Symbolic Links

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

```
1 #include <unistd.h>
2 int symlink(const char *src, const char *dst);
3 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.

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

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

18 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.

19 futimens, utimesat, and utimes Functions

- 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:

```
1 #include <sys/stat.h>
2 int futimens(int fd, const struct timespec times[2]);
3 int utimensat(int dirfd, const char *path,
4               const struct timespec times[2],
5               int flags);
6
7 struct timespec {
8     time_t tv_sec;    /* seconds */
9     long tv_nsec;    /* nanoseconds */
10 };
```

- 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.

- 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.
- 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.

```

1 #include <sys/time.h>
2 int utimes (const char *pathname,
3             const struct timeval times[2]);
4
5 struct timeval {
6     time_t tv_sec; /* seconds */
7     long tv_usec; /* microseconds */
8 };

```

- 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*).

20 mkdir, mkdirat, and rmdir Functions

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

```

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

```

- Be careful of *mode* parameter.

```

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

```

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

21 Reading Directories

- 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.

```
1 struct dirent{
2     ino_t   d_ino;
3     char    d_name[256];
4 };
5
6 #include <dirent.h>
7 DIR *opendir(const char *pathname);
8 DIR *fdopendir(int fd);
9 struct dirent *readdir(DIR *dp);
10 void rewinddir(DIR *dp);
11 int closedir(DIR *dp);
12 long telldir(DIR *dp);
13 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`).

22 chdir, fchdir, and getcwd Functions

- 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.

```
1 #include <unistd.h>
2 int chdir (const char *pathname);
3 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`).
- 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.

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

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

23 Device Special Files

- 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`.
- 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`).

24 Summary

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

The End of Chapter 4.