Files and Directories

Malikov Davronbek

Distributed Systems Lab.Gyeongsang National University

11.10.2018

1. Introduction
2. stat, fstat, fstatat, and lstat Functions
3. File Types
4. Set-User-ID and Set-Group-ID
5. File access permission
6. Ownership of New Files and Directories
7. File Size
8. link,linkat,unlink,unlinkat, and remove Function
9. File times
10. Reading Directories
11. Summary

Introduction

In the last lesson we covered the basic functions that perform I/O.The main important notation was I/O for regular files—opening a file, and reading or writing a file.We'll now look at additional features of the file system and the properties of a file.We'll start with the stat functions and go through each member of the stat structure,looking at all the attributes of a file. In this process, we'll also describe each of the functions that modify these attributes: change the owner, change the permissions, and so on.

stat, fstat, fstatat, and lstat Functions

Now We will start with four Functions stat such as , fstat, fstatat, and lstat Functions and information they return. 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, not the file referenced by the symbolic link.

stat,fstat,fstatat,and lstat Functions

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; when the AT_SYMLINK_NOFOLLOW flag is set, fstatat will not follow symbolic links, but rather returns information about the link itself.Otherwise, the default is to follow symbolic links, returning information about the file to which the symbolic link points. If the fd argument has the value AT_FDCWD and the pathname argument is a relative pathname, then fstatat evaluates the pathname argument relative to the current directory.If the pathname argument is an absolute pathname, then ft argument is ignored. In these two cases, fstatat behaves like either stat or lstat, depending on the value of flag.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.

stat,fstat,fstatat,and lstat Functions

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; Prior to the 2008 version of the standard, the time fields were named st_atime, st_mtime, and st_ctime, and were of type time_t (expressed in seconds). The timespec structure enables higher-resolution timestamps. The old names can be defined in terms of the tv_sec members for compatibility. For example, st_atime can be defined as st_atim.tv_sec.

stat,fsat,fstatat,and lstat Functions

Note that most members of the stat structure are specified by a primitive system data type.The biggest user of the stat functions is probably the ls -l command, to learn all the information about a file.

File types

We've talked about two different types of files so far: regular files and directories. Most files on a UNIX system are either regular files or directories.However it does not mean there are only two types of files becouse there are additional types of files.They types are:

1. Regular file.
2. Directory file.
3. Block special file.
4. Character special file.
5. FIFO.
6. Socket.
7. Symbolic link.

File types

1. **Regular file**. The most common type of file, which contains data of some form. There is no distinction to the UNIX kernel whether this data is text or binary.Any interpretation of the contents of a regular file is left to the application processing the file.One notable exception to this is with binary executable files. To execute a program, the kernel must understand its format. All binary executable files conform to a format that allows the kernel to identify where to load a program's text and data.
2. **Directory file**. A file that contains the names of other files and pointers to information on these files. Any process that has read permission for a directory file can read the contents of the directory, but only the kernel can write directly to a directory file. Processes must use the functions described in this chapter to make changes to a directory.

File types

3. **Block special file.** A type of file providing buffered I/O access in fixed-size units to devices such as disk drives.**Note that FreeBSD no longer supports block special files. All access to devices is through the character special interface.**
4. **Character special file**. A type of file providing unbuffered I/O access in variable-sized units to devices. All devices on a system are either block special files or character special files.
5. **FIFO.** A type of file used for communication between processes. It's sometimes called a named pipe.

File types

6. **Socket**A type of file used for network communication between processes. A socket can also be used for non-network communication between processes on a single host.
7. **Symbolic link** A type of file that points to another file. We talk more about symbolic links later.

Set-User-ID and Set-Group-ID

Every process has six or more IDs associated with it. **They are:**

1. Real User ID(Who we really are)
2. Real Group ID(Who we really are)
3. effective user ID(used for file access permission checks)
4. effective group ID(used for file access permission checks)
5. supplementry groups ID(used for file access permission checks)
6. saved-set user ID(saved by exec functions)
7. saved-set group ID(saved by exec functions)

Set-user-ID and Set-group-ID

1. The real user ID and real group ID identify who we really are. These two fields are taken from our entry in the password file when we log in. Normally, these values don't change during a login session, although there are ways for a superuser process to change them
2. The effective user ID, effective group ID, and supplementary group IDs determine our file access permissions
3. The saved set-user-ID and saved set-group-ID contain copies of the effective user ID and the effective group ID, respectively, when a program is executed. We describe the function of these two saved values.

Set-user-ID and Set-group-ID

Normally, the effective user ID equals the real user ID, and the effective group ID equals the real group ID. Every file has an owner and a group owner. The owner is specified by the st_uid member of the stat structure; the group owner, by the st_gid member. When we execute a program file, the effective user ID of the process is usually the real user ID, and the effective group ID is usually the real group ID. However, we can also set a special flag in the file's mode word (st_mode) that says, "When this file is executed, set the effective user ID of the process to be the owner of the file (st_uid)." Similarly, we can set another bit in the file's mode word that causes the effective group ID to be the group owner of the file (st_gid). These two bits in the file's mode word are called the set-user-ID bit and the set-group-ID bit.

Set-user-ID and Set-group-ID

For example, if the owner of the file is the superuser and if the file's set-user-ID bit is set, then while that program file is running as a process, it has superuser privileges. This happens regardless of the real user ID of the process that executes the file. As an example, the UNIX System program that allows anyone to change his or her password, passwd(1), is a set-user-ID program. This is required so that the program can write the new password to the password file, typically either /etc/passwd or /etc/shadow, files that should be writable only by the superuser. Because a process that is running set-user-ID to some other user usually assumes extra permissions, it must be written carefully.

Set-user-ID and Set-group-ID

Returning to the stat function, the set-user-ID bit and the 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, respectively.

File access permission

The st_mode value also encodes the access permission bits for the file. When we say file, we mean any of the file types that we described earlier. All the file types — directories, character special files, and so on—have permissions. Many people think of only regular files as having access permissions.

File access directions

1.The first rule is that whenever we want to open any type of file by name, we must have execute permission in each directory mentioned in the name, including the current directory, if it is implied. This is why the execute permission bit for a directory is often called the search bit. For example, to open the file /usr/include/stdio.h, we need execute permission in the directory /, execute permission in the directory /usr, and execute permission in the directory /usr/include. We then need appropriate permission for the file itself, depending on how we're trying to open it: read-only, read–write, and so on. If the current directory is /usr/include, then we need execute permission in the current directory to open the file stdio.h. This is an example of the current directory being implied, not specifically mentioned. It is identical to our opening the file ./stdio.h.

File access permissions

Note that read permission for a directory and execute permission for a directory mean different things. Read permission lets us read the directory, obtaining a list of all the filenames in the directory. Execute permission lets us pass through the directory when it is a component of a pathname that we are trying to access.

File access permissions

2. The read permission for a file determines whether we can open an existing file for reading: the O_RDONLY and O_RDWR flags for the open function. **3.** The write permission for a file determines whether we can open an existing file for writing: the O_WRONLY and O_RDWR flags for the open function. **4.** We must have write permission for a file to specify the O_TRUNC flag in the open function. **5.** We cannot create a new file in a directory unless we have write permission and execute permission in the directory. textbf6. delete an existing file, we need write permission and execute permission in the directory containing the file. We do not need read permission or write permission for the file itself. **7.** Execute permission for a file must be on if we want to execute the file using any of the seven exec functions.The file also has to be a regular file.

File access permissions

The file access tests that the kernel performs each time a process opens, creates, or deletes a file depend on the owners of the file (st_uid and st_gid), the effective IDs of the process (effective user ID and effective group ID), and the supplementary group IDs of the process, if supported. The two owner IDs are properties of the file, whereas the two effective IDs and the supplementary group IDs are properties of the process. The tests performed by the kernel are as follows: **1.** If the effective user ID of the process is 0 (the superuser), access is allowed. This gives the superuser free rein throughout the entire file system. **2.** If the effective user ID of the process equals the owner ID of the file (i.e., the process owns the file), access is allowed if the appropriate user access permission bit is set. Otherwise, permission is denied. By appropriate access permission bit, we mean that if the process is opening the file for reading, the user-read bit must be on. If the process is opening the file for writing, the user-write bit must be on. If the process is executing the file, the user-execute bit must be on.

File access permissions

3. If the effective group ID of the process or one of the supplementary group IDs of the process equals the group ID of the file, access is allowed if the appropriate group access permission bit is set. Otherwise, permission is denied. **4.** If the appropriate other access permission bit is set, access is allowed. Otherwise, permission is denied. ****** These four steps are tried in sequence. ****Note that** if the process owns the file (step 2), access is granted or denied based only on the user access permissions; the group permissions are never looked at. Similarly, if the process does not own the file but belongs to an appropriate group, access is granted or denied based only on the group access permissions; the other permissions are not looked at.

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: **1.** The group ID of a new file can be the effective group ID of the process. **2.** The group ID of a new file can be the group ID of the directory in which the file is being created. Using the second option—inheriting the directory's group ID—assures us that all files and directories created in that directory will have the same group ID as the directory. This group ownership of files and directories will then propagate down the hierarchy from that point. This is used in the Linux directory /var/mail, for example.

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. For a regular file, a file size of 0 is allowed. We'll get an end-of-file indication on the first read of the file. For a directory, the file size is usually a multiple of a number, such as 16 or 512. We talk about reading directories in Section 4.22. For a symbolic link, the file size is the number of bytes in the filename. For example, in the following case, the file size of 7 is the length of the pathname usr/lib: lrwxrwxrwx 1 root 7 Sep 25 07:14 lib– usr/lib **Note that symbolic links do not contain the normal C null byte at the end of the name, as the length is always specified by st_size** Most contemporary UNIX systems provide the fields s t_blksize and st_blocks. The first is the preferred block size for I/O for the file, and the latter is the actual number of 512-byte blocks that are allocated. Recall from last lessons that we encountered the minimum amount of time required to read a file when we used st_blksize for the read operations.

link,linkat,unlink,unlinkat and remove Function

We can use either the link function or the linkat function to create a link to an existing file. We can create a new directory entry, newpath, that references the existing file existingpath. If the newpath already exists, an error is returned. Only the last component of the newpath is created. The rest of the path must already exist. With the linkat function, the existing file is specified by both the efd and existingpath arguments, and the new pathname is specified by both the nfd and newpath arguments. By default, if either pathname is relative, it is evaluated relative to the corresponding file descriptor. If either file descriptor is set to AT_FDCWD, then the corresponding pathname, if it is a relative pathname, is evaluated relative to the current directory.

link,linkat,unlink,unlinkat and remove Function

If either pathname is absolute, then the corresponding file descriptor argument is ignored. When the existing file is a symbolic link, the flag argument controls whether the linkat function creates a link to the symbolic link or to the file to which the symbolic link points. If the AT_SYMLINK_FOLLOW flag is set in the flag argument, then a link is created to the target of the symbolic link. If this flag is clear, then a link is created to the symbolic link itself. The creation of the new directory entry and the increment of the link count must be an atomic operation.Most implementations require that both pathnames be on the same file system, although POSIX.1 allows an implementation to support linking across file systems. If an implementation supports the creation of hard links to directories, it is restricted to only the superuser. This constraint exists because such hard links can cause loops in the file system, which most utilities that process the file system aren't capable of handling.

link,linkat,unlink,unlinkat and remove Function

As mentioned earlier, to unlink a file, we must have write permission and execute permission in the directory containing the directory entry, as it is the directory entry that we will be removing. Also, as mentioned in Section 4.10, if the sticky bit is set in this directory we must have write permission for the directory and meet one of the following criteria:

1. own the file
2. own the directory
3. Have superusert privileges Only when the link count reaches 0 can the contents of the file be deleted. One other condition prevents the contents of a file from being deleted: as long as some process has the file open, its contents will not be deleted. When a file is closed, the kernel first checks the count of the number of processes that have the file open. If this count has reached 0, the kernel then checks the link count; if it is 0, the file's contents are deleted.

link,linkat,unlink,unlinkat and remove Function

If the pathname argument is a relative pathname, then the unlinkat function evaluates the pathname relative to the directory represented by the fd file descriptor argument. If the fd argument is set to the value AT_FDCWD, then the pathname is evaluated relative to the current working directory of the calling process. If the pathname argument is an absolute pathname, then the fd argument is ignored. The flag argument gives callers a way to change the default behavior of the unlinkat function. When the AT_REMOVEDIR flag is set, then the unlinkat function can be used to remove a directory, similar to using rmdir. If this flag is clear, then unlinkat operates like unlink.

File Times

In stat, fstat, fstatat, and lstat Functions, we discussed how the 2008 version of the Single UNIX Specification increased the resolution of the time fields in the stat structure from seconds to seconds plus nanoseconds. The actual resolution stored with each file's attributes depends on the file system implementation. For file systems that store timestamps in second granularity, the nanoseconds fields will be filled with zeros. For file systems that store timestamps in a resolution higher than seconds, the partial seconds value will be converted into nanoseconds and returned in the nanoseconds fields.

File Times

Note the difference between the modification time (st_mtim) and the changed-status time (st_ctim). The modification time indicates when the contents of the file were last modified. The changed-status time indicates when the i-node of the file was last modified. In this chapter, we've described many operations that affect the i-node without changing the actual contents of the file: changing the file access permissions, changing the user ID, changing the number of links, and so on. Because all the information in the i-node is stored separately from the actual contents of the file, we need the changed-status time, in addition to the modification time.

File Times

Note that the system does not maintain the last-access time for an i-node. This is why the functions access and stat, for example, don't change any of the three times. The access time is often used by system administrators to delete files that have not been accessed for a certain amount of time. The classic example is the removal of files named a.out or core that haven't been accessed in the past week. The find(1) command is often used for this type of operation. The modification time and the changed-status time can be used to archive only those files that have had their contents modified or their i-node modified. The ls command displays or sorts only on one of the three time values. By default, when invoked with either the -l or the -t option, it uses the modification time of a file. The -u option causes the ls command to use the access time, and the -c option causes it to use the changed-status time.

Reading Directories

Directories can be read by anyone who has access permission to read the directory. But only the kernel can write to a directory, to preserve file system sanity. Recall from File access permissions that the write permission bits and execute permission bits for a directory determine if we can create new files in the directory and remove files from the directory — they don't specify if we can write to the directory itself. The actual format of a directory depends on the UNIX System implementation and the design of the file system. Earlier systems, such as Version 7, had a simple structure: each directory entry was 16 bytes, with 14 bytes for the filename and 2 bytes for the i-node number. When longer filenames were added to 4.2BSD, each entry became variable length, which means that any program that reads a directory is now system dependent. To simplify the process of reading a directory, a set of directory routines were developed and are part of POSIX.1.

Reading Directories

Many implementations prevent applications from using the read function to access the contents of directories, thereby further isolating applications from the implementation-specific details of directory formats.

Reading Directories

The fdopendir function first appeared in version 4 of the Single UNIX Specification. It provides a way to convert an open file descriptor into a DIR structure for use by the directory handling functions. The telldir and seekdir functions are not part of the base POSIX.1 standard. They are included in the XSI option in the Single UNIX Specification, so all conforming UNIX System implementations are expected to provide them. Recall our use of several of these functions in the program shown in Figure 1.3, our bare-bones implementation of the ls command. The dirent structure defined in ¡dirent.h¿ is implementation dependent. Implementations define the structure to contain at least the following two members:

Reading Directories

1. ino_t d_ino; /* i-node number */
2. char d_name[]; /* null-terminated filename */ **The d_ino entry** is not defined by POSIX.1, because it is an implementation feature, but it is defined as part of the XSI option in POSIX.1. POSIX.1 defines only the d_name entry in this structure. **Note that** the size of the d_name entry isn't specified, but it is guaranteed to hold at least NAME_MAX characters, not including the terminating null byte Since the filename is null terminated, however, it doesn't matter how d_name is defined in the header, because the array size doesn't indicate the length of the filename.

Summary

In the Files and Directories has centered on the stat function. We've gone through each member in the stat structure in detail. This, in turn, led us to examine all the attributes of UNIX files and directories. We've looked at how files and directories might be laid out in a file system, and we've seen how to navigate the file system namespace. A thorough understanding of all the properties of files and directories and all the functions that operate on them is essential to UNIX programming.