Files and Directories

Advanced Programming in the UNIX Environment

Chun-Ying Huang <chuang@cs.nctu.edu.tw>

Outline

Introduction

File Information

File Permissions

File Systems

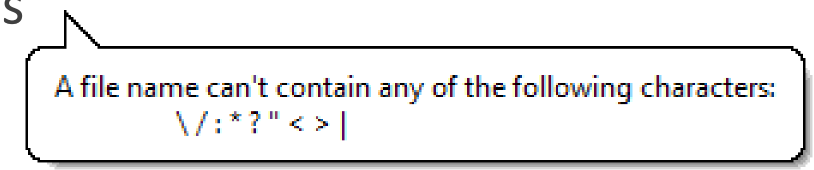
Directory Operations

Device Special Files

Filename

Any character except slash (`/`) and null (`\0`)

- Compared to Windows



Usually up to 255 characters (the `PATH_MAX` constant)

Every directory contains two filenames

- Single dot (.), and
- Double dots (. .)
- Even in the root directory

Filenames are independent of language encodings

- It is interpreted by the user terminal

Pathname

A sequence of one or more filenames

Absolute pathname – with a leading slash

- `/usr/bin/gcc`

Relative pathname – without a leading slash or with a dot

- Suppose the current working directory is /usr
- All the followings points to the same file
  - `bin/gcc`
  - `./bin/gcc`
  - `../usr/bin/gcc`

Directory

Working Directory

- Every process has a working directory
- Also called the *current working directory* (cwd)
- Can be changed using the `chdir` function

Home Directory

- The initial working directory when a user logged in
- Obtained from the `/etc/passwd` file

```
root:x:0:0:root:/root:/bin/bash
chuang:x:1000:1000:Chun-Ying Huang,,,:/home/chuang:/bin/bash
```

File Information

stat, fstat, And lstat(2) Functions

Return information about a file

Synopsis

- `int stat(const char *path, struct stat *buf);`
- `int fstat(int fd, struct stat *buf);`
- `int lstat(const char *path, struct stat *buf);`
- Returns: 0 if OK, -1 on error

`stat` and `lstat` are basically equivalent, except `lstat` does not follow a symbolic link

- It returns the information of the symbolic link itself

`fstat` do the same thing for an opened file

Common File Information

File type and permissions

Number of hard links

User ID and group ID

Device number (of the containing file system)

Device number for special files

File size

Block size and number of used blocks

Timestamps: access, modification, and change

The stat Structure

```
struct stat {
    dev_t      st_dev;       /* ID of device containing file */
    ino_t      st_ino;       /* inode number */
    mode_t     st_mode;      /* protection */
    nlink_t    st_nlink;     /* number of hard links */
    uid_t      st_uid;       /* user ID of owner */
    gid_t      st_gid;       /* group ID of owner */
    dev_t      st_rdev;      /* device ID (if special file) */
    off_t      st_size;      /* total size, in bytes */
    blksize_t st_blksize;    /* blocksize for filesystem I/O */
    blkcnt_t  st_blocks;     /* number of 512B blocks allocated */
    time_t     st_atime;     /* time of last access */
    time_t     st_mtime;     /* time of last modification */
    time_t     st_ctiem;     /* time of last status change */
};
```

File Types

Regular file

Directory file

Block special file

Character special file

FIFO

Socket

Symbolic link

File Permissions

UIDs and GIDs Associated with a Process

Real user ID and real group ID

- UID and GID – Who we really are
- Taken from the password file when login

Effective user ID and effective group ID

- EUID and EGID
- Used for file access permission checks

Saved set-user-ID and saved set-group-ID

- SUID and SGID
- Saved by the `exec` function
- The effective user ID and effective group ID when a program is executed

Relationships Between UIDs/GIDs

Normally, the EUID equals the UID and the EGID equals the GID

EUID and EGID can be set by the running program using `setuid(2)` and `setgid(2)`, respectively

Special permission set with SUID and SGID

- If a program *P* owned by *UserA* has a SUID permission
- Any one who executes the program *P* will be automatically `setuid` to *UserA*
- SGID does similar to SUID, but is applied to group id
- Only root is able to set SUID/SGID permissions for a file

Applications of SUID and SGID

An example – The `passwd` program

- The program used to change user's password

```
$ ls -la /usr/bin/passwd
-rwsr-xr-x 1 root root 32988 2008-12-08 17:17 /usr/bin/passwd
```

- Changing a user password requires to modify the `/etc/shadow` file, which is only accessible to the superuser
- With the SUID permission, a user is able to run the `passwd` program with root's permission (EUID=0)

File Access Permissions (1/3)

A file is always associated with a user ID (the owner) and a group ID

9-bit permissions

- user-read, user-write, user-execute
- group-read, group-write, group-execute
- other-read, other-write, other-execute
- Permissions are usually represented in octal
  - For example, 0755
  - Convert 0755 to binary: `111` `101` `101`

```
$ ls -la /bin/bash
-rwxr-xr-x 1 root root 725136 2008-05-13 02:48 /bin/bash
```

File Access Permissions (2/3)

Rules

- We can only access files in a directory with valid execute permissions
- We can open a file for read if we have valid read permissions
- We can open a file for write and truncate if we have valid write permissions
- To delete an existing file in a directory, we only need valid write and execute permissions of the directory
- An executable must have valid execute permissions

File Access Permissions (3/3)

File access test precedence

- If the EUID of the process is 0, access is allowed
- If the EUID of the process equals the owner ID of the file, access is allowed if the appropriate user access permission bits are set
- If the EGID or supplementary GIDs of the process equals the group ID, access is allowed if the appropriate group access permission bits are set
- If the appropriate other access permission bits are set, access is allowed

Access is allowed if one of the above checks passes

Ownership of New Files and Directories

The user ID of a new file is set to the EUID of the creating process

The group ID of a new file can be set by …

- The EGID of the creating process, or
- The GID of the parent directory

The choice of group ID depends on the OS …

- FreeBSD 5.2.1/Mac OS X 10.3 – by the parent directory
- Linux – depends on a mount option (`grpid`)
  - If `grpid` is set or directory has SGID – by the parent directory
  - Otherwise – by the EGID of the creating process

The access(2) Function

Check accessibility of a file

From a real user's perspective

Synopsis

- `int access(const char *path, int mode);`
- Returns: 0 if OK, -1 on error

The mode can be the bitwise OR of the following constants

- `R_OK, W_OK, X_OK` – Test for read, write, and execute permissions
- `F_OK` – Test for the existence of the file

An Example for access(2) Function

```
#include "apue.h"
#include <fcntl.h>

int main(int argc, char *argv[]) {
        if (argc != 2)
                err_quit("usage: a.out <pathname>");
        if (access(argv[1], R_OK) < 0)
                err_ret("access error for %s", argv[1]);
        else
                printf("read access OK\n");
        if (open(argv[1], O_RDONLY) < 0)
                err_ret("open error for %s", argv[1]);
        else
                printf("open for reading OK\n");
        exit(0);
}
```

An Example for access(2) Function (Cont'd)

```
$ ls -l a.out
-rwxrwxr-x 1 sar 15945 Nov 30 12:10 a.out
$ ./a.out a.out
read access OK
open for reading OK
$ ls -l /etc/shadow
-r-------- 1 root 1315 Jul 17 2002 /etc/shadow
$ ./a.out /etc/shadow
access error for /etc/shadow: Permission denied
open error for /etc/shadow: Permission denied
$ sudo su -                          become superuser
Password:                            enter superuser password
# chown root a.out                   change file's user ID to root
# chmod u+s a.out                         and turn on set-user-ID bit
# ls -l a.out                        check owner and SUID bit
-rwsrwxr-x 1 root 15945 Nov 30 12:10 a.out
# exit                               go back to normal user
$ ./a.out /etc/shadow
access error for /etc/shadow: Permission denied
open for reading OK
```

The umask(2) Function

Sets the file mode creation mask for the process

Synopsis

- `mode_t umask(mode_t cmask);`
- Returns: previous file mode creation mask

Changing the umask of a process doesn't affect the umask of its parent

See examples in the next page

The umask(2) Function (Cont'd)

```
#define RWRWRW (S_IRUSR|S_IWUSR|S_IRGRP|S_IWGRP|S_IROTH|S_IWOTH)
int main(void) {
        umask(0);
        if (creat("foo", RWRWRW) < 0)
                err_sys("creat error for foo");
        umask(S_IRGRP | S_IWGRP | S_IROTH | S_IWOTH);
        if (creat("bar", RWRWRW) < 0)
                err_sys("creat error for bar");
}
```

```
$ umask                  first print the current file mode creation mask
002
$ ./a.out
$ ls -l foo bar
-rw------- 1 sar 0 Dec 7 21:20 bar
-rw-rw-rw- 1 sar 0 Dec 7 21:20 foo
$ umask                  see if the file mode creation mask changed
002
```

Functions to Set File Modes and Ownerships

File modes

- `int chmod(const char *path, mode_t mode);`
- `int fchmod(int fd, mode_t mode);`

File ownerships

- `int chown(const char *path, uid_t owner, gid_t group);`
- `int fchown(int fd, uid_t owner, gid_t group);`
- `int lchown(const char *path, uid_t owner, gid_t group);`
- Note: `lchown` will not follow symbolic links

The Sticky Bit

Can be used on an executable or a directory

For an executable with the sticky bit

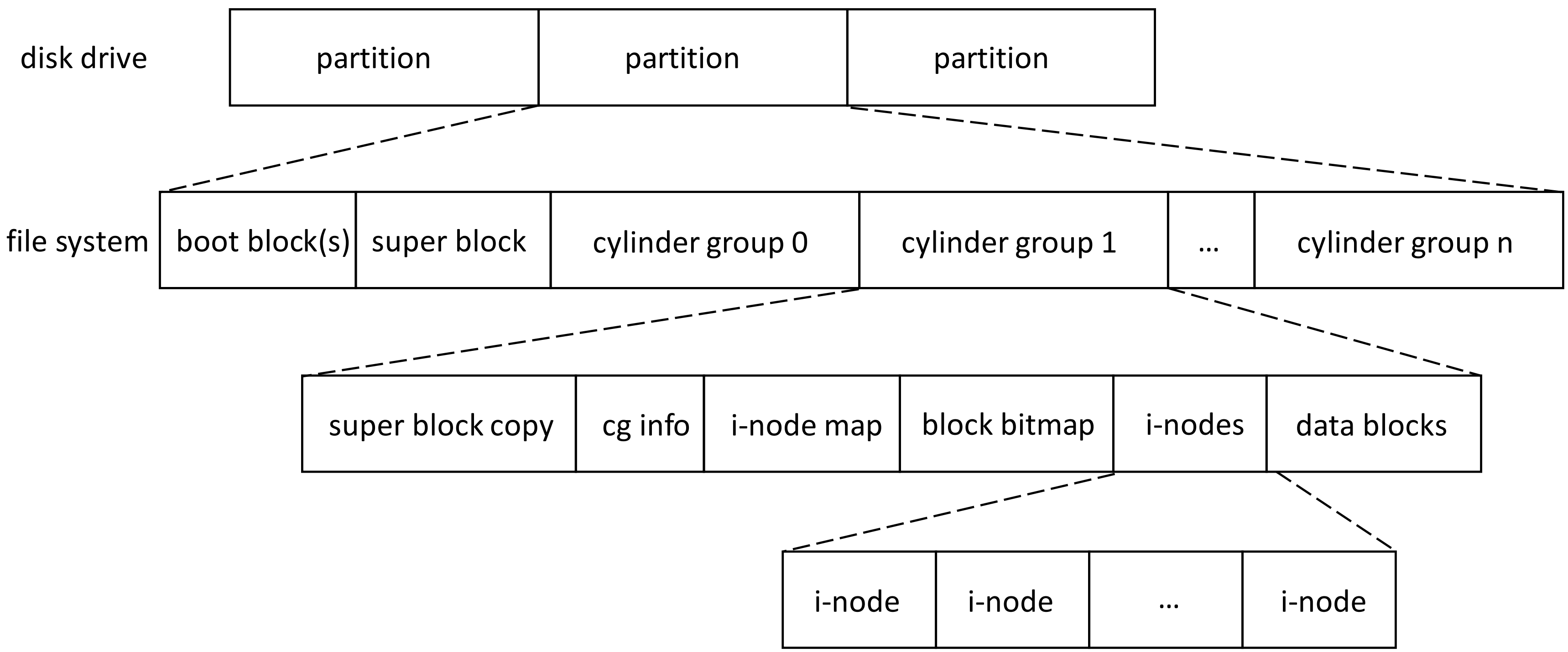
- Cache the executable in swap area after execution
- Increase performance of loading the executable

For a directory with the sticky bit

- A file in the directory can be only deleted or renamed by …
  - The user owns the file
  - The user owns the directory
  - The superuser
- Usually set for global accessible directories, such as /tmp

File Systems

Disk Drives, Partitions, and File System



Cylinder Group's i-nodes and Data Blocks

i-node – describe (meta) information about a file

- type, permission, data blocks, timestamps, reference counts, ..., etc.

i-node are often indexed using a positive integer number

Some i-node numbers has special purpose

- 0 – reserved, or does not exist
- 1 – list of bad/defective blocks
- 2 – root directory of a partition

Reference Counts

The number of pointers that points to an i-node

Common file operations to work with reference counts

- Link, unlink, and remove

Usually a newly created file has a reference count of 1

- Pointed by the containing directory

A reference count increases on being link(2)’ed

- Create a *hard-link*
- Hard links must reside in *the same* partition

A reference count decreases on being unlink(2)’ed

link, unlink, remove, and rename

Synopsis

- `int link(const char *existingpath, const char *newpath);`
- `int unlink(const char *pathname);`
- `int remove(const char *pathname);`
- `int rename(const char *oldname, const char *newname);`
- Returns: 0 if OK, -1 on error

Relevant Commands

- `ls -l` # long listing format, check the reference count
- `ls -ls` # show file sizes in 1KB blocks
- `ls -li` # show i-node numbers

Symbolic links

It is also called soft-links (in contrast to hard-links)

The ln(1) command

The size of a symbolic link is the length of its target's name

```
$ mkdir foo                          make a new directory
$ touch foo/a                        create a zero-length file
$ ln -s ../foo foo/testdir           create a symbolic link
$ ls -l foo
total 0
-rw-r----- 1 sar 0 Jan 22 00:16 a
lrwxrwxrwx 1 sar 6 Jan 22 00:16 testdir -> ../foo
```

In the above example, the foo/testdir symbolic link causes a loop!

Symbolic Links (Cont'd)

Symbolic links is able to point to an nonexistence file

```
$ ln -s /no/such/file myfile    create a symbolic link
$ ls myfile
myfile                          ls says it's there
$ cat myfile                    so we try to look at it
cat: myfile: No such file or directory
$ ls -l myfile                  try -l option
lrwxrwxrwx 1 sar 13 Jan 22 00:26 myfile -> /no/such/file
```

Treatments of Symbolic Links by Various Functions

Function	Not follow symbolic link	Follow symbolic link	Function	Not follow symbolic link	Follow symbolic link
access		•	open		•
chdir		•	opendir		•
chmod		•	pathconf		•
chown		•	readlink	•	
creat		•	remove	•	
exec		•	rename	•	
lchown	•		stat		•
link		•	truncate		•
lstat	•		unlink	•	

symlink and readlink

Synopsis

- `int symlink(const char *actualpath, const char *sympath);`
- Returns: 0 if OK, -1 on error
- `ssize_t readlink(const char *path, char *buf, size_t bufsize);`
- Returns: number of bytes placed in the buffer, -1 on error

symlink create a symbolic link *sympath* points to the *actualpath*

- *sympath* and *actualpath* need not reside in the same file system

readlink reads the targeted pathname of the given symbolic link *path*

- The function combines *open*, *read*, and *close*.

File Times

Field	Description	Example	ls(1) option
st_atime	last access time	read	-u
st_mtime	last modification time (file content)	write	default
st_ctime	last change time (i-node)	chmod, chown	-c

Effect of various functions on the access, modification, and change times

Function	Referenced file or dir			Parent directory			Function	Referenced file or dir			Parent directory		
	a	m	c	a	m	c		a	m	c	a	m	c
[f]chmod			•				pipe	•	•	•			
[f]chown			•				read	•					
creat (new)	•	•	•		•	•	remove (file)			•		•	•
creat (trunc)		•	•				remove (dir)					•	•
exec	•						rename			•		•	•
lchown			•				rmdir					•	•
link			•		•	•	[f]truncate		•	•			
mkdir	•	•	•		•	•	unlink			•		•	•
mkfifo	•	•	•		•	•	utime	•	•	•			
open (new)	•	•	•		•	•	write		•	•			
open (trunc)		•	•										

utime Function

Synopsis

- `int utime(const char *filename, const struct utimbuf *times);`
- Returns: 0 if OK, -1 on error

Time utimbuf data structure

```
struct utimbuf {
        time_t actime;
        time_t modtime;
};
```

Change the access time and modification time of a file

If *times* is *NULL*, the access time and modification time is set to the current time

Directory Operations

mkdir and rmdir Functions

Synopsis

- `int mkdir(const char *pathname, mode_t mode);`
- `int rmdir(const char *pathname);`
- Returns: 0 if OK, -1 on error

Create or remove a directory

When removing a directory …

- If a directory is not empty, the function fails
- If a directory is empty and no process has the directory open, it is removed and freed
- If a directory is empty, but a process has opened the directory, it is removed but not freed
  - No new file can be created in the to-be-removed directory
  - It is freed when the process close the directory

Reading Directories

Access permissions for directories

- Read: must have read and execute permission
- Create/write: must have write and execute permission

Synopsis

- `DIR *opendir(const char *name);`
- Returns: pointer to the directory if OK, NULL on error
- `struct dirent *readdir(DIR *dir);`
- Returns: pointer to a dirent structure if OK, NULL on reaching *EOF* or error
- `int closedir(DIR *dir);`
- Returns: 0 if OK, -1 on error

readdir Function

The dirent structure

```
struct dirent {
        ino_t          d_ino;        /* inode number */
        off_t          d_off;        /* offset to the next dirent */
        unsigned short d_reclen;     /* length of this record */
        unsigned char  d_type;       /* type of file */
        char           d_name[256];  /* filename */
};
```

The data returned by readdir() may be overwritten by subsequent calls to readdir() for the same directory stream.

Seek in an Opened Directory

Synopsis

- `void rewinddir(DIR *dir);`
- `off_t telldir(DIR *dir);`
- Returns: current location of the opened dir
- `void seekdir(DIR *dir, off_t offset);`

rewinddir resets the position of the directory stream to the beginning

seekdir set the location in the directory stream

A Sample Program – Filename Enumeration

Enumerate all files and directories in a given directory

Usage

- `./prog-1.3 {directory_name}`

```
#include "mylib.h"
#include <dirent.h>

int
main(int argc, char *argv[]) {
        DIR *dp;
        struct dirent *dirp;
        if(argc != 2)
                err_quit("usage: ls directory_name\n");
        if((dp = opendir(argv[1])) == NULL)
                err_sys("cannot open %s\n", argv[1]);
        while((dirp = readdir(dp)) != NULL)
                printf("%s\n", dirp->d_name);
        closedir(dp);
        return(0);
}
```

chdir, fchdir, and getcwd Functions

Every *process* has a current working directory
- The current working directory is inherited from the parent
- The current working directory for each process is independent

The default working directory for a user is his home directory (configured in the /etc/passwd file)

Current working directory can be changed

Synopsis
- `int chdir(const char *path);`
- `int fchdir(int fd);`
- Returns: 0 if OK, 1 on error
- `char *getcwd(char *buf, size_t size);`
- Returns: *buf* if OK, NULL on error

```
$ pwd
/usr/lib
$ mycd
chdir to /tmp succeeded
$ pwd
/usr/lib
```

Device Special Files

Device Special Files

Every file system is known by its major and minor device numbers, *encoded* in the system type dev_t

- The major number identifies the device driver
- The minor number identifies the specific sub device
- The major number can be extracted by the macro major(dev_t)
- The minor number can be extracted by the macro minor(dev_t)

Recall the stat(2) function, which returns a file status structure

- The st_dev and the st_rdev value, what's the difference?
- Let's see an example

An Example of Retrieving Device Numbers

```
int main(int argc, char *argv[]) {
    int i;
    struct stat buf;
    for (i = 1; i < argc; i++) {
        printf("%s: ", argv[i]);
        if (stat(argv[i], &buf) < 0) {
            err_ret("stat error");
            continue;
        }
        printf("dev=%d/%d", major(buf.st_dev), minor(buf.st_dev));
        if (S_ISCHR(buf.st_mode) || S_ISBLK(buf.st_mode)) {
            printf(" (%s) rdev = %d/%d",
                (S_ISCHR(buf.st_mode)) ? "character" : "block",
                major(buf.st_rdev), minor(buf.st_rdev));
        }
        printf("\n");
    }
    exit(0);
}
```

st_dev and st_rdev

Every file has a *st_dev* number, which indicates the container file system

Only device files have *st_rdev* numbers, which indicates the device/sub-device

Running the example in the previous slide

```
$ ./fig4.25-devrdev / /dev /home/chuang /dev/tty0 /dev/sda2
/: dev = 8/2
/dev: dev = 0/14
/home/chuang: dev = 8/2
/dev/tty0: dev = 0/14 (character) rdev = 4/0
/dev/sda2: dev = 0/14 (block) rdev = 8/2
```

The /dev File System

Stores device special files

It can be a real file system

- Each device special file is created with the *mknod(1)* command

It can be a pseudo file system

- Device special files are automatically generated when a device driver is registered

```
$ ls -la /dev/tty0 /dev/sda* /dev/null
crw-rw-rw- 1 root root 1, 3 2008-12-04 10:02 /dev/null
brw-rw---- 1 root disk 8, 0 2009-02-09 18:39 /dev/sda
brw-rw---- 1 root disk 8, 1 2009-02-09 18:39 /dev/sda1
brw-rw---- 1 root disk 8, 2 2009-02-09 10:39 /dev/sda2
crw-rw---- 1 root root 4, 0 2009-02-09 18:39 /dev/tty0
```

Common Device Special Files

/dev/hda*N*? – IDE disks and partitions

/dev/sda*N?* – SCSI or SATA disks and partitions

/dev/scd*N* – CD/DVD-ROMs

/dev/tty*N* – terminals

/dev/ttyS*N* – COM ports

/dev/pts/*N* – pseudo terminals

/dev/null

/dev/zero

/dev/random

Q & A