IBM

developerWorks®

Learn Linux, 101: Manage file permissions and ownership

Setting the right security on your files

Ian Shields
Senior Programmer
IBM

30 November 2010

Learn to manage file ownership and permissions on your Linux® filesystems. Learn about access modes such as suid, sgid, and the sticky bit and how to use them to enhance security. You can use the material in this article to study for the LPI 101 exam for Linux system administrator certification, or just to learn about file ownership, permissions, and security.

View more content in this series

Overview

In this article, learn to control file access through correct use of file and directory permissions and ownerships. Learn to:

- Manage access permissions on both regular and special files as well as directories
- Maintain security using access modes such as suid, sgid, and the sticky bit
- Change the file creation mask
- Grant file access to group members

Unless otherwise noted, the examples in this article use Fedora 13 with a 2.6.34 kernel. Your results on other systems may differ.

This article helps you prepare for Objective 104.5 in Topic 104 of the Linux Professional Institute's Junior Level Administration (LPIC-1) exam 101. The objective has a weight of 3.

We introduced some of the file and group ownership concepts of this article in our previous article "Learn Linux 101: Manage disk quotas." This article will help you understand those concepts more completely.

About this series

This series of articles helps you learn Linux system administration tasks. You can also use the material in these articles to prepare for Linux Professional Institute Certification level 1 (LPIC-1) exams.

See our developerWorks roadmap for LPIC-1 for a description of and link to each article in this series. The roadmap is in progress and reflects the latest (April 2009) objectives for the LPIC-1 exams: as we complete articles, we add them to the roadmap. In the meantime, you can find earlier versions of similar material, supporting previous LPIC-1 objectives prior to April 2009, in our LPI certification exam prep tutorials.

Prerequisites

To get the most from the articles in this series, you should have a basic knowledge of Linux and a working Linux system on which you can practice the commands covered in this article. Sometimes different versions of a program will format output differently, so your results may not always look exactly like the listings and figures shown here.

User and groups

Connect with Ian

Ian is one of our most popular and prolific authors. Browse all of Ian's articles on developerWorks. Check out Ian's profile and connect with him, other authors, and fellow readers in My developerWorks.

By now, you know that Linux is a multiuser system and that each user belongs to one *primary* group and possibly additional groups. It is also possible to log in as one user and become another user using the `su` or `sudo -s` commands. Ownership of files in Linux and access authority are closely related to user ids and groups, so let's review some basic user and group information.

Who am I?

If you have not become another user, your id is still the one you used to log in. If you have become another user, your prompt may include your user id, as most of the examples in this article do. If your prompt does not include your user id, then you can use the `whoami` command to check your current effective id. Listing 1 shows some examples where the prompt strings (from the PS1 environment variable) are different from the other examples in this article. Having your id in the prompt string can be a useful feature.

Listing 1. Determining effective user id

```
/home/ian$ whoami
tom
/home/ian$ exit
exit
$ whoami
ian
```

What groups am I in?

Similarly, you can find out what groups you are in by using the `groups` command. You can find out both user and group information using the `id` command. Add a user id parameter to either `groups` or `id` to see information for that user id instead of the current user id. See Listing 2 for some examples. Note that without a userid, the `id` command will also display SELinux context as well as basic id information.

Listing 2. Determining group membership

```
[ian@echidna ~]$ id
uid=1000(ian) gid=1000(ian) groups=1000(ian),505(development),8093(editor)
context=unconfined_u:unconfined_r:unconfined_t:s0-s0:c0.c1023
[ian@echidna ~]$ id ian
uid=1000(ian) gid=1000(ian) groups=1000(ian),8093(editor),505(development)
[ian@echidna ~]$ groups
ian development editor
[ian@echidna ~]$ id tom
uid=1012(tom) gid=1012(tom) groups=1012(tom),505(development)
[ian@echidna ~]$ groups tom
tom : tom development
[ian@echidna ~]$ su tom
Password:
[tom@echidna ian]$ groups
tom development
[tom@echidna ian]$ groups ian
ian : ian editor development
```

File ownership and permissions

Just as every user has an id and is a member of one primary group, so every file on a Linux system has one owner and one group associated with it.

Ordinary files

Use the `ls -l` command to display the owner and group.

Listing 3. Determining file ownership

```
[ian@echidna ~]$ ls -l /bin/bash .bashrc helloworld.C
-rw-r--r--. 1 ian  ian            124 Mar 31  2010 .bashrc
-rwxr-xr-x. 1 root root        943360 May 21  2010 /bin/bash
-rw-rw-r--. 1 ian  development    116 Nov 30 10:21 helloworld.C
```

In this particular example, user ian's .bashrc file is owned by him and is in the ian group, which is his primary group. Similarly, /bin/bash is owned by user root and is in the group root. However, helloworld.C is owned by user ian, but its group is development. User names and groups names come from separate namespaces, so a given name may be both a user name and a group name. In fact, many distributions default to creating a matching group for each new user.

The Linux permission model has three types of permission for each filesystem object. The permissions are read (r), write (w), and execute (x). Write permission includes the ability to alter or delete an object. In addition, these permissions are specified separately for the file's owner, members of the file's group, and everyone else.

Referring back to the first column of Listing 3, notice that it contains an eleven-character string. The eleventh character is a recent addition. We'll discuss it in a moment. The first character describes the type of object (`-` for an ordinary file in this example) and the remaining nine characters represent three groups of three characters. The first group indicates the read, write, and execute permissions for the file's owner. A `-` indicates that the corresponding permission is not granted. So user ian can read and write the .bashrc file, but not execute it; while root can read, write, **and** execute the /bin/bash file. The second group indicates the read, write, and execute permissions for the file's group. Members of the development group can read or write ian's helloworld.C file, while everyone else can only read it. Similarly, members of the root group and everyone else can read or execute the /bin/bash file.

Directories

Directories use the same permissions flags as regular files, but they are interpreted differently. Read permission for a directory allows a user with that permission to list the contents of the directory. Write permission means a user with that permission can create or delete files in the directory. Execute permission allows the user to enter the directory and access any subdirectories. Without execute permission, the filesystem objects inside a directory are not accessible. Without read permission, the filesystem objects inside a directory are not viewable in a directory listing, but these objects can still be accessed as long as you know the full path to the object on disk. Listing 4 is a very artificial example to illustrate these points.

Listing 4. Permissions and directories

```
[ian@echidna ~]$ ls -l /home
total 32
drwxr-x---. 38 editor   editor      12288 Nov 30 10:49 editor
drwxr-x---.  4 greg     development  4096 Nov 30 12:44 greg
drwx------. 21 gretchen gretchen     4096 Nov 30 11:26 gretchen
drwxr-xr-x. 41 ian      ian          4096 Nov 30 10:51 ian
drwx------. 21 ianadmin ianadmin     4096 May 28  2010 ianadmin
d-wx--x--x. 21 tom      tom          4096 Nov 30 11:30 tom
[ian@echidna ~]$ ls -a ~greg/.ba*
/home/greg/.bash_history  /home/greg/.bash_profile
/home/greg/.bash_logout   /home/greg/.bashrc
[ian@echidna ~]$ ls -a ~gretchen
ls: cannot open directory /home/gretchen: Permission denied
[ian@echidna ~]$ ls -a ~tom
ls: cannot open directory /home/tom: Permission denied
[ian@echidna ~]$ head -n 3 ~tom/.bashrc
# .bashrc

# Source global definitions
```

The first character of a long listing describes the type of object (`d` for a directory). User greg's home directory has read and execute permission for members of the development group, so users tom and ian can list the directory. User gretchen's home directory has neither read nor execute permission for the gretchen group or other users, so user ian cannot access it. User tom's home has execute but not read permission, so user ian cannot list the contents, but can access objects within the directory if he knows they exist.

Other filesystem objects

The output from `ls -l` may contain filesystem objects other than files and directories as shown by the first character in the listing. We will see more of these later in this article, but for now, note the possible types of objects.

Table 1. Filesystem object types

Code	Object type
-	Regular file
d	Directory
l	Symbolic link
c	Character special device
b	Block special device
p	FIFO
s	Socket

The eleventh character

The eleventh character in a long listing from the `ls` command is a recent enhancement, so some distributions may still show only the first ten characters. In other cases, the eleventh character is a space, so you may not notice it. This character specifies whether an alternate access method applies to the file. When the character following the file mode bits is a space, there is no alternate access method. When it is a printing character, then there is such a method. The method might be an access control list for example. GNU `ls` uses a '.' (dot) character to signify a file with only an SELinux security context. Files with any other combination of alternate access methods are marked with the '+' (plus) character.

Changing permissions

Adding permissions

Suppose you create a "Hello world" shell script. When you first create the script, it will usually not be executable. Use the `chmod` command with the `+x` option to add the execute permissions as shown in Listing 5.

Listing 5. Creating an executable shell script

```
[ian@echidna ~]$ echo 'echo "Hello world!"'>hello.sh
[ian@echidna ~]$ ls -l hello.sh
-rw-rw-r--. 1 ian ian 20 Nov 30 13:05 hello.sh
[ian@echidna ~]$ ./hello.sh
bash: ./hello.sh: Permission denied
[ian@echidna ~]$ chmod +x hello.sh
[ian@echidna ~]$ ./hello.sh
Hello world!
[ian@echidna ~]$ ls -l hello.sh
-rwxrwxr-x. 1 ian ian 20 Nov 30 13:05 hello.sh
```

You can use `+r` to set the read permissions, and `+w` to set the write permissions in a similar manner. In fact, you can use any combination of `r`, `w`, and `x` together. For example, using `chmod +rwx` would set all the read, write, and execute permissions for a file. This form of `chmod` adds permissions that are not already set.

Being selective

You may have noticed in the above example that execute permission was set for the owner, group, **and** others. To be more selective, you may prefix the mode expression with `u` to set the permission for users, `g` to set it for groups, and `o` to set it for others. Specifying `a` sets the permission for all users, which is equivalent to omitting it. Listing 6 shows how to add user and group write and execute permissions to another copy of the shell script.

Listing 6. Selectively adding permissions

```
[ian@echidna ~]$ echo 'echo "Hello world!"'>hello2.sh
[ian@echidna ~]$ chmod ug+xw hello2.sh
[ian@echidna ~]$ ls -l hello2.sh
-rwxrwxr--. 1 ian ian 20 Nov 30 13:08 hello2.sh
```

Removing permissions

Sometimes you need to remove permissions rather than add them. Simply change the `+` to a `-`, and you remove any of the specified permissions that are set. Listing 7 shows how to remove all permissions for other users on the two shell scripts.

Listing 7. Removing permissions

```
[ian@echidna ~]$ ls -l hello*.sh
-rwxrwxr--. 1 ian ian 20 Nov 30 13:08 hello2.sh
-rwxrwxr-x. 1 ian ian 20 Nov 30 13:05 hello.sh
[ian@echidna ~]$ chmod o-xrw hello*.sh
[ian@echidna ~]$ ls -l hello*.sh
-rwxrwx---. 1 ian ian 20 Nov 30 13:08 hello2.sh
-rwxrwx---. 1 ian ian 20 Nov 30 13:05 hello.sh
```

Note that you can change permissions on more than one file at a time. As with some other commands you met in the articles for topic 103, you can even use the `-R` (or `--recursive`) option to operate recursively on directories and files.

Setting permissions

Now that you can add or remove permissions, you may wonder how to set just a specific set of permissions. Do this using `=` instead of `+` or `-`. To set the permissions on the above scripts so that other users have no access rights, you could use `chmod o= hello*`, instead of the command we used to remove permissions.

If you want to set different permissions for user, group, or other, you can separate different expressions by commas; for example, `ug=rwx,o=rx`, or you can use numeric permissions, which are described next.

Octal permissions

So far you have used symbols (ugoa and rxw) to specify permissions. There are three possible permissions in each group. You can also set permissions using octal numbers instead of symbols. Permissions set in this way use up to four octal digits. We will look at the first digit when we discuss attributes. The second digit defines user permissions, the third group permissions and the fourth other permissions. Each of these three digits is constructed by adding the desired permissions settings: read (4), write (2), and execute (1). In the example for hello.sh in Listing 5, the script was created with permissions -rw-r--r--, corresponding to octal 644. Setting execute permission for everyone changed the mode to 755.

Using numeric permissions is very handy when you want to set all the permissions at once without giving the same permissions to each of the groups. Use Table 2 as a handy reference for octal permissions.

Table 2. Numeric permissions

Symbolic	Octal
`rwx`	`7`
`rw-`	`6`
`r-x`	`5`
`r--`	`4`
`-wx`	`3`
`-w-`	`2`
`--x`	`1`
`---`	`0`

Access modes

When you log in, the new shell process runs with your user and group ids. These are the permissions that govern your access to any files on the system. This usually means that you cannot access files belonging to others and cannot access system files. In fact, we as users are totally dependent on other programs to perform operations on our behalf. Because the programs you start inherit *your* user id, they cannot access any filesystem objects for which you haven't been granted access.

An important example is the /etc/passwd file, which cannot be changed by normal users directly, because write permission is enabled only for root: However, normal users need to be able to modify /etc/passwd somehow, whenever they need to change their password. So, if the user is unable to modify this file, how can this be done?

suid and sgid

The Linux permissions model has two special access modes called suid (set user id) and sgid (set group id). When an executable program has the suid access modes set, it will run as if it had been started by the file's owner, rather than by the user who really started it. Similarly, with the sgid access modes set, the program will run as if the initiating user belonged to the file's group rather than to his own group. Either or both access modes may be set.

Listing 8 shows that the `passwd` executable is owned by root:

Listing 8. suid access mode on /usr/bin/passwd

```
[ian@echidna ~]$ ls -l /usr/bin/passwd
-rwsr-xr-x. 1 root root 34368 Apr  6  2010 /usr/bin/passwd
```

Note that in place of an `x` in the user's permission triplet, there's an `s`. This indicates that, for this particular program, the suid and executable bits are set. So when `passwd` runs, it will execute as if the root user had launched it with full superuser access, rather than that of the user who ran it. Because `passwd` runs with `root` access, it can modify /etc/passwd.

The suid and sgid bits occupy the same space as the `x` bits for user and group in a long directory listing. If the file is executable, the suid or sgid bits, if set, will be displayed as lowercase `s`, otherwise they are displayed as uppercase `S`.

While suid and sgid are handy, and even necessary in many circumstances, improper use of these access mode can allow breaches of a system's security. You should have as few suid programs as possible. The `passwd` command is one of the few that **must** be suid.

Setting suid and sgid

The suid and sgid bits are set and reset symbolically using the letter `s`; for example, `u+s` sets the suid access mode, and `g-s` removes the sgid mode. In the octal format, suid has the value 4 in the first (high order) digit, while sgid has the value 2.

Directories and sgid

When a directory has the sgid mode enabled, any files or directories created in it will inherit the group id of the directory. This is particularly useful for directory trees that are used by a group of people working on the same project. Listing 9 shows how user greg could set up a directory that all users of the development group could use, along with an example of how user gretchen could create a file in the directory. As created, the file gretchen.txt allows groups members to write to the file, so gretchen uses **`chmod g-w`** to disable group write capability.

Listing 9. sgid access mode and directories

```
[greg@echidna ~]$ mkdir lpi101
[greg@echidna ~]$ chmod g+ws lpi101
[greg@echidna ~]$ ls -ld lpi101
drwxrwsr-x. 2 greg development 4096 Nov 30 13:30 lpi101/
[greg@echidna ~]$ su - gretchen
Password:
[gretchen@echidna ~]$ touch ~greg/lpi101/gretchen.txt
[gretchen@echidna ~]$ ls -l ~greg/lpi101/gretchen.txt
-rw-rw-r--. 1 gretchen development 0 Nov 30 14:12 /home/greg/lpi101/gretchen.txt
[gretchen@echidna ~]$ chmod g-w ~greg/lpi101/gretchen.txt
[gretchen@echidna ~]$ ls -l ~greg/lpi101/gretchen.txt
-rw-r--r--. 1 gretchen development 0 Nov 30 14:12 /home/greg/lpi101/gretchen.txt
```

Any member of the development group can now create files in user greg's lpi101 directory. As Listing 10 shows, other members of the group cannot update the file gretchen.txt. However, they do have write permission to the directory and can therefore delete the file.

Listing 10. sgid access mode and file ownership

```
[gretchen@echidna ~]$ su - tom
Password:
[tom@echidna ~]$ echo "something" >> ~greg/lpi101/gretchen.txt
-bash: /home/greg/lpi101/gretchen.txt: Permission denied
[tom@echidna ~]$ rm ~greg/lpi101/gretchen.txt
rm: remove write-protected regular empty file `/home/greg/lpi101/gretchen.txt'? y
[tom@echidna ~]$ ls -l ~greg/lpi101/
total 0
```

The sticky bit

You have just seen how anyone with write permission to a directory can delete files in it. This might be acceptable for a workgroup project, but is not desirable for globally shared file space such as the /tmp directory. Fortunately, there is a solution.

The remaining access mode bit is called the *sticky* bit. It is represented symbolically by `t` and numerically as a 1 in the high-order octal digit. It is displayed in a long directory listing in the place of the executable flag for other users (the last character), with the same meaning for upper and lower case as for suid and sgid. If set for a directory, it permits only the owning user or the superuser (root) to delete or unlink a file. Listing 11 shows how user greg could set the sticky bit on his lpi101 directory and also shows that this bit is set for /tmp.

Listing 11. Sticky directories

```
[greg@echidna ~]$ chmod +t lpi101
[greg@echidna ~]$ ls -ld lpi101 /tmp
drwxrwsr-t.  2 greg development  4096 Nov 30 14:16 lpi101
drwxrwxrwt. 24 root root        12288 Nov 30 14:22 /tmp
```

On a historical note, UNIX® systems used to use the sticky bit on files to hoard executable files in swap space and avoid reloading. Modern Linux kernels ignore the sticky bit if it is set for files.

Access mode summary

Table 3 summarizes the symbolic and octal representation for the three access modes discussed here.

Table 3. Access modes

Access mode	Symbolic	Octal
suid	`s` with `u`	`4000`
sgid	`s` with `g`	`2000`
sticky	`t`	`1000`

Combining this with the earlier permission information, you can see that the full octal representation corresponding to greg's lpi101 permissions and access modes of drwxrwsr-t is 3775. While the `ls` command does not display the octal permissions, you can display them using the `find` command as shown in Listing 12

Listing 12. Printing symbolic and octal permissions

```
[greg@echidna ~]$ find . -name lpi101  -printf "%M %m %f\n"
drwxrwsr-t 3775 lpi101
```

Immutable files

The access modes and permissions provide extensive control over who can do what with files and directories. However, they do not prevent things such as inadvertent deletion of files by the root user. Although beyond the scope of LPI Topic 104.5, there are some additional *attributes* available on various filesystems that provide additional capabilities. One of these is the *immutable* attribute. If this is set, even root cannot delete the file until the attribute is unset.

Use the `lsattr` command to see whether the immutable flag (or any other attribute) is set for a file or directory. To make a file immutable, use the `chattr` command with the `-i` flag.

Listing 13 shows that user root can create an immutable file but cannot delete it until the immutable flag is removed.

Listing 13. Immutable files

```
[root@echidna ~]# touch keep.me
[root@echidna ~]# chattr +i keep.me
[root@echidna ~]# lsattr keep.me
----i--------e- keep.me
[root@echidna ~]# rm -f keep.me
rm: cannot remove `keep.me': Operation not permitted
[root@echidna ~]# chattr -i keep.me
[root@echidna ~]# rm -f keep.me
```

Changing the immutable flag requires root authority, or at least the CAP_LINUX_IMMUTABLE capability. Making files immutable is often done as part of a security or intrusion detection effort. See the capabilities man page (`man capabilities`) for more information.

The file creation mask

When a new file is created, the creation process specifies the permissions that the new file should have. Often, the mode requested is 0666, which makes the file readable and writable by anyone. Directories usually default to 0777. However, this permissive creation is affected by a *umask* value, which specifies what permissions a user does **not** want to grant automatically to newly created files or directories. The system uses the umask value to reduce the originally requested permissions. You can view your umask setting with the `umask` command, as shown in Listing 14.

Listing 14. Displaying octal umask

```
[ian@echidna ~]$ umask
0002
```

Remember that the umask specifies which permissions should **not** be granted. On Linux systems, where users do not have private groups the umask normally defaults to 0022, which **removes** group and other write permission from new files. Where users have a private group (as on the Fedora system used in these examples) the umask normally defaults to 0002 which removes the write permission for other users. Use the `-S` option to display the umask symbolically, in a form that shows which are the permissions that **are** allowed.

Use the `umask` command to set a umask as well as display one. So, if you would like to keep your files more private and disallow all group or other access to newly created files, you would use a umask value of 0077. Or set it symbolically using `umask u=rwx,g=,o=`, as illustrated in Listing 15.

Listing 15. Setting the umask

```
[ian@echidna ~]$ umask -S
u=rwx,g=rwx,o=rx
[ian@echidna ~]$ umask u=rwx,g=,o=
[ian@echidna ~]$ umask
0077
[ian@echidna ~]$ touch newfile
[ian@echidna ~]$ ls -l newfile
-rw-------. 1 ian ian 0 Nov 30 15:40 newfile
```

Setting file owner and group

File group

To change the group of a file, use the `chgrp` command with a group name and one or more filenames. You may also use the group number if you prefer. An ordinary user must own the file and also be a member of the group to which the file's group is being changed. The root user may change files to any group. Listing 16 shows an example.

Listing 16. Changing group ownership

```
[ian@echidna ~]$ touch file{1,2}
[ian@echidna ~]$ ls -l file*
-rw-rw-r--. 1 ian ian 0 Nov 30 15:54 file1
-rw-rw-r--. 1 ian ian 0 Nov 30 15:54 file2
[ian@echidna ~]$ chgrp development file1
[ian@echidna ~]$ chgrp 505 file2
[ian@echidna ~]$ ls -l file*
-rw-rw-r--. 1 ian development 0 Nov 30 15:54 file1
-rw-rw-r--. 1 ian development 0 Nov 30 15:54 file2
```

As with many of the commands covered in this tutorial, `chgrp` has a `-R` option to allow changes to be applied recursively to all selected files and subdirectories.

Default group

When you studied Access modes above, you learned how setting the sgid mode on a directory causes new files created in that directory to belong to the group of the directory rather than to the group of the user creating the file.

You may also use the `newgrp` command to temporarily change your primary group to another group of which you are a member. A new shell will be created, and when you exit the shell, your previous group will be reinstated, as shown in Listing 17.

Listing 17. Using newgrp to temporarily change default group

```
[ian@echidna ~]$ groups
ian development editor
[ian@echidna ~]$ newgrp development
[ian@echidna ~]$ groups
development ian editor
[ian@echidna ~]$ touch file3
[ian@echidna ~]$ ls -l file3
-rw-r--r--. 1 ian development 0 Nov 30 16:00 file3
[ian@echidna ~]$ exit
[ian@echidna ~]$ groups
ian development editor
```

File owner

The root user can change the ownership of a file using the `chown` command. In its simplest form, the syntax is like the `chgrp` command, except that a user name or numeric id is used instead of a group name or id. The file's group may be changed at the same time by adding a colon and a group name or id right after the user name or id. If only a colon is given, then the user's default group is used. Naturally, the `-R` option will apply the change recursively. Listing 18 shows an example.

Listing 18. Using chown to changing file ownership

```
[ian@echidna ~]$ touch file4
[ian@echidna ~]$ su -
Password:
[root@echidna ~]# ls -l ~ian/file4
-rw-rw-r--. 1 ian ian 0 Nov 30 16:04 /home/ian/file4
[root@echidna ~]# chown greg ~ian/file4
[root@echidna ~]# ls -l ~ian/file4
-rw-rw-r--. 1 greg ian 0 Nov 30 16:04 /home/ian/file4
[root@echidna ~]# chown tom:gretchen ~ian/file4
[root@echidna ~]# ls -l ~ian/file4
-rw-rw-r--. 1 tom gretchen 0 Nov 30 16:04 /home/ian/file4
[root@echidna ~]# chown :tom ~ian/file4
[root@echidna ~]# ls -l ~ian/file4
-rw-rw-r--. 1 tom tom 0 Nov 30 16:04 /home/ian/file4
```

An older form of specifying both user and group used a dot instead of a colon. This is no longer recommended as it may cause confusion when names include a dot.

This completes your introduction to file and directory permissions on Linux.

Resources

Learn

- Develop and deploy your next app on the IBM Bluemix cloud platform.
- Use the developerWorks roadmap for LPIC-1 to find the developerWorks articles to help you study for LPIC-1 certification based on the April 2009 objectives.
- At the LPIC Program site, find detailed objectives, task lists, and sample questions for the three levels of the Linux Professional Institute's Linux system administration certification. In particular, see their April 2009 objectives for LPI exam 101 and LPI exam 102. Always refer to the LPIC Program site for the latest objectives.
- Review the entire LPI exam prep series on developerWorks to learn Linux fundamentals and prepare for system administrator certification based on earlier LPI exam objectives prior to April 2009.
- The Directory listing chapter of the GNU Coreutils manual explains the parts of directory listings.
- The Linux Documentation Project has a variety of useful documents, especially its HOWTOs.
- In the developerWorks Linux zone, find hundreds of how-to articles and tutorials, as well as downloads, discussion forums, and a wealth of other resources for Linux developers and administrators.
- Stay current with developerWorks technical events and webcasts focused on a variety of IBM products and IT industry topics.
- Attend a free developerWorks Live! briefing to get up-to-speed quickly on IBM products and tools, as well as IT industry trends.
- Watch developerWorks on-demand demos ranging from product installation and setup demos for beginners, to advanced functionality for experienced developers.
- Follow developerWorks on Twitter, or subscribe to a feed of Linux tweets on developerWorks.

Get products and technologies

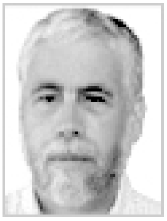
- Evaluate IBM products in the way that suits you best: Download a product trial, try a product online, use a product in a cloud environment, or spend a few hours in the SOA Sandbox learning how to implement Service Oriented Architecture efficiently.

Discuss

- Participate in the discussion forum for this content.
- Get involved in the My developerWorks community. Connect with other developerWorks users while exploring the developer-driven blogs, forums, groups, and wikis.

About the author

Ian Shields



Ian Shields works on a multitude of Linux projects for the developerWorks Linux zone. He is a Senior Programmer at IBM at the Research Triangle Park, NC. He joined IBM in Canberra, Australia, as a Systems Engineer in 1973, and has since worked on communications systems and pervasive computing in Montreal, Canada, and RTP, NC. He has several patents and has published several papers. His undergraduate degree is in pure mathematics and philosophy from the Australian National University. He has an M.S. and Ph.D. in computer science from North Carolina State University. Learn more about Ian in Ian's profile on developerWorks Community.