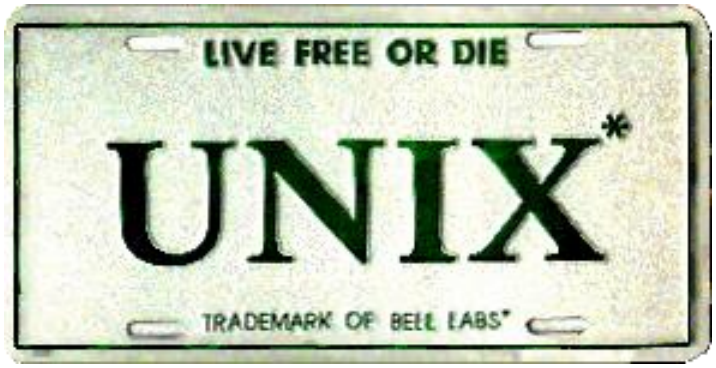


UNIX for Programmers and Users

“UNIX for Programmers and Users”
Third Edition, Prentice-Hall, GRAHAM GLASS, KING ABLES

Change File Permissions: chmod

chmod -R change fileName

- The chmod utility changes the modes (permissions) of the specified files according to the change parameters, which may take the following forms:

clusterSelection+newPermissions (add permissions)
clusterSelection-newPermissions (subtract permissions)
clusterSelection=newPermissions (assign permissions absolutely)

- where clusterSelection is any combination of:
  - u (user/owner)
  - g (group)
  - o (others)
  - a (all)

  and newPermissions is any combination of
  - r (read)
  - w (write)
  - x (execute)
  - s (set user ID/set group ID)

Changing File Permissions

- Note that changing a directory's permission settings doesn't change the settings of the files that it contains.

- The **-R** option recursively changes the modes of the files in directories.

Ex: Remove read permission from groups

$ ls -lg heart.final --> to view the settings before the change.

```
-rw-r-----   1    glass     music  213  Jan  31  00:12  heart.final
```

$ chmod g-r heart.final

$ ls -lg heart.final

```
-rw-------   1    glass     music  213  Jan  31  00:12  heart.final
```

$ _

Changing File Permissions: examples

Requirement	Change parameters
Add group write permission	g+w
Remove user read and write permission	u-rw
Add execute permission for user, group, and others.	a+x
Give the group read permission only.	g=r
Add write permission for user, and remove group read permission.	u+w,g-r

Changing File Permission: examples

- Example:

```
$ cd              --> change to home directory.
$ ls  -ld   .              --> list attributes of home directory.
drwxr-xr-x  45   glass   4096  Apr   29   14:35
$ chmod   o-rx                  --> update permissions.
$ ls  -ld   .                   --> confirm.
drwxr-x---    45   glass   4096  Apr  29  14:35
$ _
```

Changing File Permissions Using Octal Numbers

- The chmod utility allows to specify the new permission setting of a file as an octal number.
- Each octal digit represents a permission triplet.

For example, for a file to have the permission settings of rwxr-x---

the octal permission setting would be 750, calculated as follows:

	User	Group	Others
setting	rwx	r-x	---
binary	111	101	000
octal	7	5	0

Changing File Permissions Using Octal Numbers

- The octal permission setting would be supplied to chmod as follows:

```
$ chmod  750   .                    --> update permissions.

$ ls  -ld  .                    --> confirm.
drwxr-x---      45   glass     4096    Apr  29 14:35
$ _
```

Listing Group: groups

- groups

- The groups utility allows you to list all of the groups that you're a member of, and it works like this:

groups userId

- When invoked with no arguments, the group utility displays a list of all of the groups that you are a member of.

- If the name of a user is specified, a list of the groups to which that user belongs are displayed.

- Example:

```
$ groups userId                    --> list my groups
cs          music
$ _
```

Changing File Group: chgrp

- Changing a File's group : chgrp

chgrp -R groupname fileName

- The chgrp utility allows a user to change the group of files that he/she owns.
- A super-user can change the group of any file.
- All of the files that follow the groupname argument are affected.
- **The -R option** recursively changes the group of the files in a directory.

Changing File Group: example

```
$ ls -lg heart.final
-rw-r--r--   1   glass   cs   213  Jan  31  00:12  heart.final

$ chgrp  music heart.final      --> change the group.

$ ls -lg heart.final       --> confirm the changes.
-rw-r--r--    1    glass    music  213  Jan  31  00:12  heart.final
$ _
```

- The chgrp utility is also used to change the group of a directory.

Changing File Owner: chown

chown -R newUserId fileName

- The chown utility allows a super-user to change the ownership of files.
- Some Unix versions allow the owner of the file to reassign ownership to another user.
- All of the files that follow the newUserId argument are affected.
- The -R option recursively changes the owner of the files in directories.

Changing File Owner: chown

- Example: change the ownership of “heart.final” to “tim” and then back to “glass” again:

$ ls -lg heart.final --> to view the owner before the change.
-rw-r----- 1 glass music 213 Jan 31 00:12 heart.final

$ chown tim heart.final --> change the owner to “tim”.

$ ls -lg heart.final --> to view the owner after the change.
-rw-r----- 1 tim music 213 Jan 31 00:12 heart.final

$ chown glass heart.final --> change the owner back to “glass”.
$ _

Change User Groups: newgrp

newgrp [-][groupname]

- The newgrp utility with a groupname as an argument, creates a new shell with an effective group ID corresponding to the groupname.
- The old shell sleeps until the termination of the newly created shell.
- User must be a member of the specified group.
- If the argument is a dash(-) instead of a groupname, a shell is created with the same settings as those of the shell that was created by logging into the system.

Changing Groups: example

$ date > test1 --> create from a “cs” group shell.

$ newgrp music --> create a “music” group shell.

$ date > test2 --> create from a “music” group shell.
^D

$ ls -lg test1 test2 --> look at each file’s attributes.

```
-rw-r--r--    1   glass   cs      29  Jan  31   22:57   test1
-rw-r--r--    1   glass   music   29  Jan  31   22:57   test2
$ _
```

Adding group & assigning to user

- sudo groupadd -g 2000 ug1
- sudo tail /etc/group
- groups srd
- sudo adduser srd ug1
- groups srd

UNIX Shells

- **INTRODUCTION**

  A shell is a program that is an interface between a user and the raw operating system.

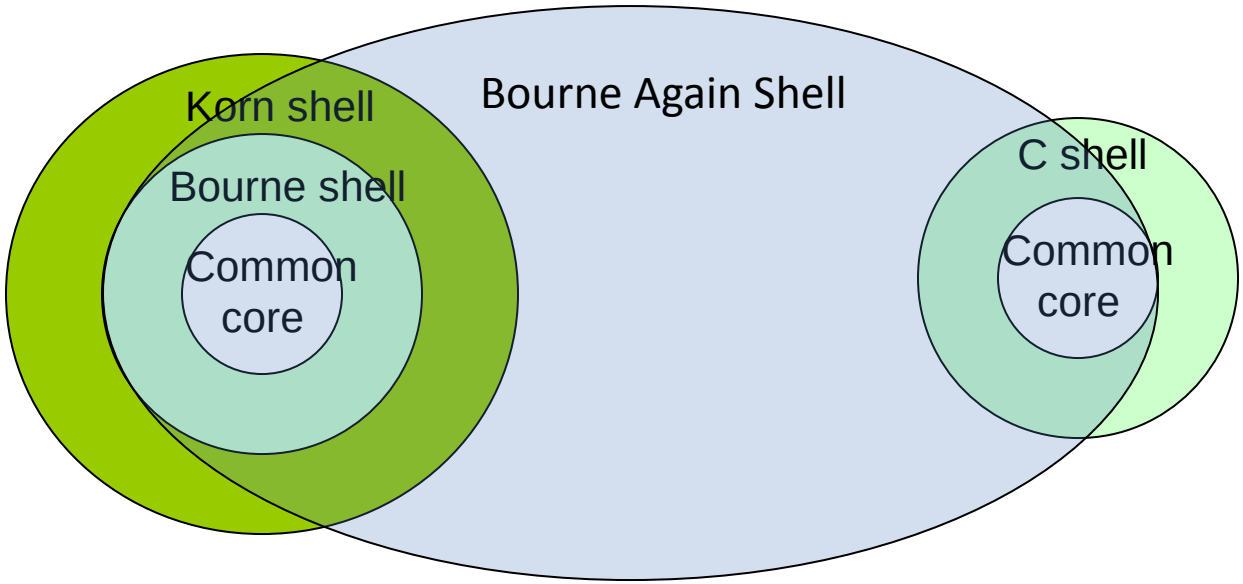
  It makes basic facilities such as multitasking and piping easy to use, and it adds useful file-specific features such as wildcards and I/O redirection.

  There are four common shells in use:

  - the Bourne shell
  - the Korn shell
  - the C shell
  - the Bash shell (Bourne Again Shell)

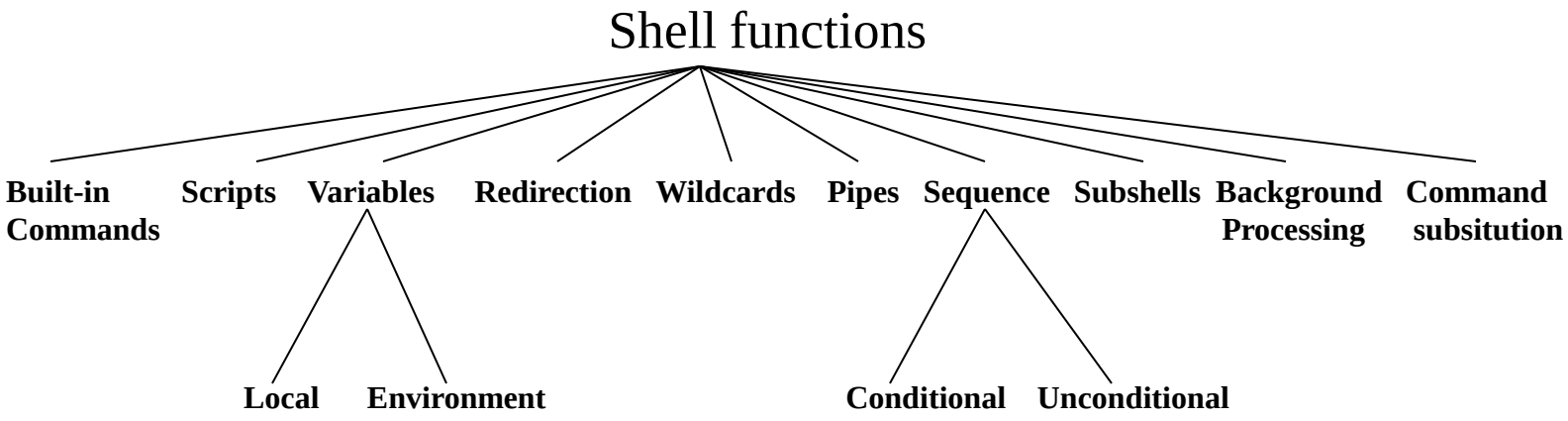
- **SHELL FUNCTIONALITY**

  - Here is a diagram that illustrates the relationship among the four shells:



- **SHELL FUNCTIONALITY**

  - The features shared by the four shells



- **SELECTING A SHELL**

The system administrator chooses a shell for any UNIX user.

$ prompt represents probably a Bash, Bourne or a Korn shell.
% prompt represents probably a C shell.

- **Utility : chsh**
  - **chsh** allows you to change your default login shell.
    It prompts you for the full pathname of the new shell,
    which is then used as your shell for subsequent logins.

  - In order to use **chsh**, you must know the full pathnames of the four shells. Here they are:

Shell	Full pathname
Bourne	/bin/sh
Bash	/bin/bash
Korn	/bin/ksh
C	/bin/csh

- **SELECTING A SHELL**
  Change the default login shell from a Bourne shell to a Bash shell:

```
$ echo  $SHELL          ---> display the name of current login shell.
  /bin/bash                ---> full pathname of the Bash shell.


$ chsh                        ---> change the login shell from bash to sh.
Changing login shell for glass
Old shell :  /bin/bash      ---> pathname of old shell is displayed.
New shell: /bin/sh  ---> enter full pathname of new shell.


$ echo  $SHELL
  /bin/sh                ---> full pathname of the Bourne shell.
$
```

- **SHELL OPERATIONS**

When a shell is invoked, either automatically during a login or manually from a keyboard or script, it follows a preset sequence:

1. It reads a special startup file, typically located in the user's home directory, that contains some initialization information.

2. It displays a prompt and waits for a user command.

3. If the user enters a Control-D character on a line of its own, this command is interpreted by the shell as meaning "end of input", and it causes the shell to terminate;

   otherwise, the shell executes the user's command and returns to step 2.

- **SHELL OPERATIONS**

Commands range from simple utility invocations like:

```
$ ls
```

to complex-looking pipeline sequences like:

```
$ ps -ef  | sort  | wc -l
```

- a command with a backslash(\) character, and the shell will allow you to continue the command on the next line:

```
$ echo this is a very long shell command and needs to \
   be extended with the line-continuation character.  Note \
   that a single command may be extended for several lines.
$ _
```

- **EXECUTABLE FILES VERSUS BUILT-IN COMMANDS**

  Most UNIX commands invoke utility programs that are stored in the directory hierarchy.

  Utilities are stored in files that have execute permission.

  For example, when you type

  $ ls

  the shell locates the executable program called “ls”, which is typically found in the “/bin” directory, and executes it.

- **Displaying Information : echo**

The built-in echo command displays its arguments to standard output and works like this:

> Shell Command: **echo** arg
>
> echo is a built-in shell command that displays all of its arguments to standard output.
> By default, it appends a new line to the output.

- **Changing Directories : cd**

The built-in cd command changes the current working directory of the shell to a new location.

- **METACHARACTERS**

Some characters are processed specially by a shell and are known as metacharacters.

All four shells share a core set of common metacharacters, whose meanings are described in next two slides.

- **METACHARACTERS**

Symbol	Meaning
>	Output redirection; writes standard output to a file.
>>	Output redirection; appends standard output to a file.
<	Input redirection; reads standard input from a file.
<<	Input redirection; End of file (EoF) customization.
*	File-substitution wildcard; matches zero or more characters.
?	File-substitution wildcard; matches any single character.
[…]	File-substitution wildcard; matches any character between the brackets.

- **METACHARACTERS**

Symbol	Meaning
|	Pipe symbol; sends the output of one process to the input of another.
||	Conditional execution; executes a command if the previous one fails.
&&	Conditional execution; executes a command if the previous one succeeds.
&	Runs a command in the background.
$	Expands the value of a variable.
\	Prevents special interpretation of the next character.

- **METACHARACTERS**

- When you enter a command,
  the shell scans it for metacharacters and processes them specially.

  When all metacharacters have been processed,
  the command is finally executed.

- **Backslash(\)**

To turn off the special meaning of a metacharacter,
precede it by a backslash(\) character.

Here's an example:

```
$ echo  hi > file   --->  store output of echo in "file".

$ cat  file          --->  look at the contents of "file".
  hi

$ echo hi \> file   --->  inhibit > metacharacter.
  hi > file            --->   > is treated like other characters.

$ _
```

- **Redirection**

The shell redirection facility allows to:

1) store the output of a process to a file (output redirection)
2) use the contents of a file as input to a process (input redirection)

Output redirection

To redirect output, use either the “>” or “>>” metacharacters.

The sequence

```
$ command > fileName
```

sends the standard output of command to the file with name fileName.

The shell creates the file with name fileName if it doesn’t already exist or overwrites its previous contents if it does already exist.

- If the file already exists but doesn't have write permission, an error occurs.

```
$ cat > alice.txt              ---> creates a text file.
In my dreams that fill the night,
I see your eyes,
^D                              ---> end of input.

$ cat  alice.txt
In my dreams that fill the night, ---> look at its contents.
I see your eyes,
$ _
```

- The sequence

    $ command >> fileName

appends the standard output of command to the file with name fileName.

$ cat >> alice.txt ---> append to the file.
And I fall into them,
Like Alice fell into Wonderland.
^D ---> end of input.

$ cat alice.txt ---> look at the new contents.
In my dreams that fill the night,
I see your eyes,
And I fall into them,
Like Alice fell into Wonderland.
$ _

- The Bash, C and Korn shells also provide protection against accidental overwriting of a file due to output redirection.

In Bash:

```
$ set -o noclobber

$ echo text > test

$ echo text > test
bash: test: cannot overwrite existing file

$ echo text >| test
$ _
```

set +o noclobber → to revert the effect

- **Input Redirection**

To redirect input, use either the `<` or `<<` metacharacters.

The sequence

```
$ command < fileName
```

executes command using the contents of the file fileName as its standard input.

If the file doesn't exist or doesn't have read permission, an error occurs.

- When the shell encounters a sequence of the form

$ command << word

- it copies its standard input up to, but not including,
the line starting with word into a buffer and then executes command using the contents of the buffer as its standard input.

- that allows shell programs(scripts) to supply the standard input to other commands as in-line text,

```
$ cat << eof
> line 1
> line 2
> line 3
> eof
line 1
line 2
line 3
$ _
```

- **FILENAME SUBSTITUTION(WILDCARDS)**

- All shells support a wildcard facility that allows you to select files that satisfy a particular name pattern from the file system.

- The wildcards and their meanings are as follows:

Wildcard	Meaning
*	Matches any string, including the empty string.
?	Matches any single character.
[..]	Matches any one of the characters between the brackets. A range of characters may be specified by separating a pair of characters by a hyphen.

- Prevent the shell from processing the wildcards in a string by surrounding the string with single quotes(apostrophes) or double quotes.

- A backslash(/) character in a filename must be matched explicitly.

$ ls -FR ---> recursively list the current directory.

```
a.c        b.c        cc.c          dir1/              dir2/

dir1:
d.c        e.e

dir2:
f.d        g.c
```

$ ls *.c ---> list any text ending in ".c".

```
a.c        b.c        cc.c
```

$ ls ?.c ---> list text for which one character is followed by ".c".

```
a.c        b.c
```

```
$ ls [ac]*          ---> list any string beginning with "a" or "c".
a.c        cc.c

$ ls [A-Za-z]*   ---> list any string beginning with a letter.
a.c        b.c          cc.c

$ ls dir*/*.c      ---> list all files ending with ".c" in "dir*" directories
                   ---> ( that is, in any directories beginning with "dir" ).
dir1/d.c     dir2/g.c


$ ls */*.c          ---> list all files ending in ".c" in any subdirectory.
dir1/d.c     dir2/g.c

$ ls *2/?.? ?.?  ---> list all files with extensions in "2*" directories
                                  and current directory.
a.c      b.c      dir2/f.d        dir2/g.c
$ _
```