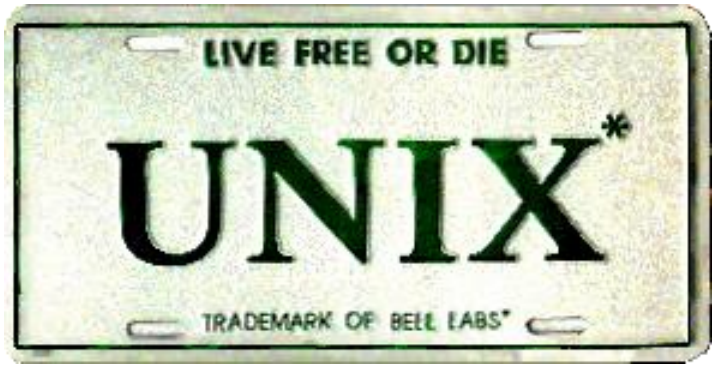


UNIX Shells

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

Slides adapted from Prof. Andrzej (AJ) Bieszczad

UNIX Shells

- **INTRODUCTION**

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

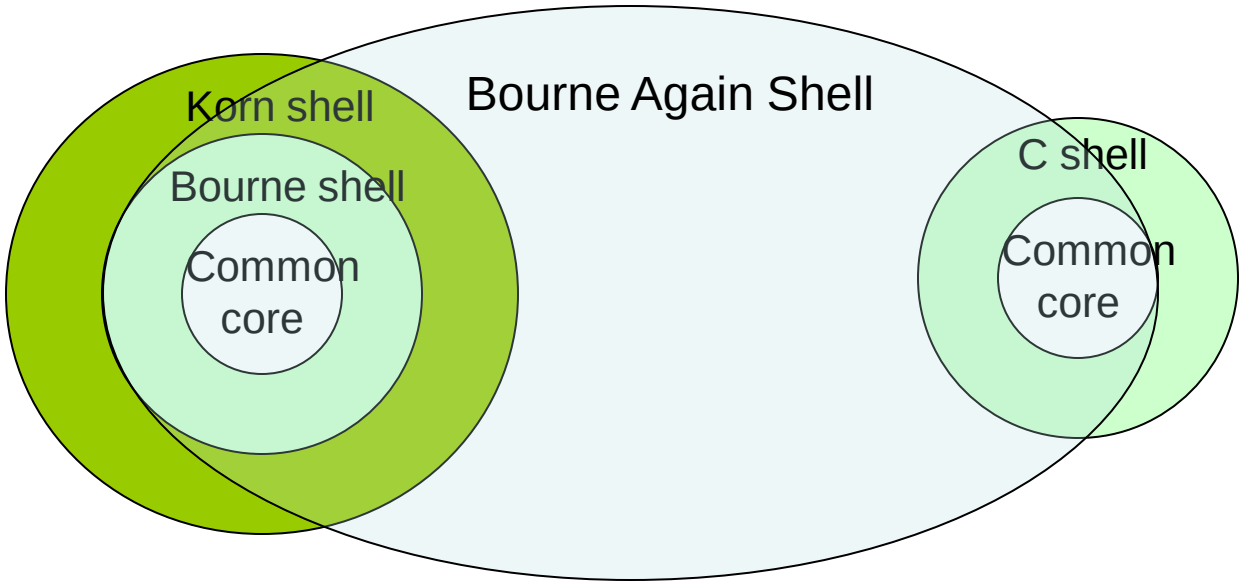
It makes basic facilities such as multitasking and piping easy to use, and it adds useful file-specific features such as wildcards and I/O redirection.

There are four common shells in use:

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

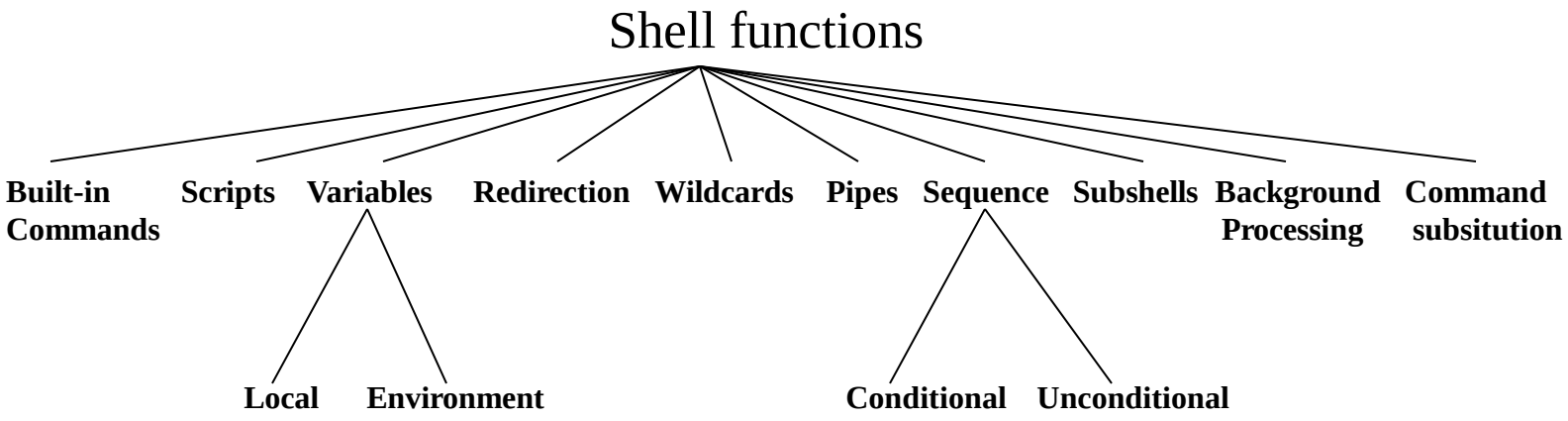
- **SHELL FUNCTIONALITY**

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



- **SHELL FUNCTIONALITY**

  - The features shared by the four shells



UNIX 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

UNIX Shells

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

```
login : glass              ---> log in.
password :                 ---> secret.
$ echo  $SHELL             ---> display the name of current login shell.
  /bin/sh                  ---> full pathname of the Bourne shell.

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

$ ^D                       ---> terminate login shell.
login : glass              ---> log back in again.
password :                 ---> secret.
$ echo  $SHELL
  /bin/bash                ---> full pathname of the Bash 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  | ul  -tdumb  | lp
```

- 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 as follow:

- **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.
*	File-substitution wildcard; matches zero or more characters.
?	File-substitution wildcard; matches any single character.
[…]	File-substitution wildcard; matches any character between the brackets.

UNIX Shells

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.

UNIX Shells

- 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.

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.

UNIX Shells

- 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,
$ _
```

UNIX Shells

- 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.
$ _
```

UNIX Shells

- 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
$ _
```

- **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.

UNIX Shells

- 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.

UNIX Shells

- 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
```

UNIX Shells

```
$ 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 in ".c" files 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
$ _
```

UNIX Shells

- **PIPES**

- Shells allow you to use the standard output of one process as the standard input of another process by connecting the processes together using the pipe(|) metacharacter.

- The sequence

```
$ command1 | command2
```

causes the standard output of command1 to "flow through" to the standard input of command2.

- Any number of commands may be connected by pipes.

- A list of commands in this way is called a *pipeline*.

- Based on one of the basic UNIX philosophies: large problems can often be solved by a chain of smaller processes

UNIX Shells

- Example, pipe the output of the **ls** utility to the input of the **wc** utility in order to count the number of files in the current directory.

```
$ ls            ---> list the current directory.
a.c   b.c    cc.c   dir1   dir2

$ ls | wc -w
    5

$ ls -l | awk '{ print $1 }' | sort        ---> example
```

- **COMMAND SUBSTITUTION**

A command surrounded by grave accents (`) - back quote - is executed, and its standard output is inserted in the command's place in the entire command line.

Any new lines in the output are replaced by spaces.

For example:

```
$ echo  the date today is `date`
the date today is Wednesday August 24 11:40:55 2016
$ _
```

UNIX Shells

- By piping the output of who to the wc utility,
  it's possible to count the number of users on the system:

```
$ who   ---> look at the output of who.
posey       ttyp0        Jan   22  15:31  ( blackfoot:0.0 )
glass       ttyp3        Feb    3  00:41  ( bridge05.utdalla )
huynh       ttyp5        Jan   10  10:39  ( atlas.utdallas.e )

$ echo  there  are  `who  |  wc -l`   users on the system
there are 3 users on the system
$ _
```

- **SEQUENCES**

If you enter a series of simple commands or pipelines separated by semicolons, the shell will execute them in sequence, from left to right.

This facility is useful for type-ahead(and think-ahead) addicts who like to specify an entire sequence of actions at once.

Here's an example:

```
$ date;  pwd;   ls    ---> execute three commands in sequence.
Wednesday August 24 11:40:55 2016
/home/glass/wild
a.c    b.c    cc.c       dir1       dir2
$ _
```

- Each command in a sequence may be individually I/O redirected as well:

```
$ date > date.txt;  ls;   pwd  > pwd.txt
a.c       b.c       cc.c       date.txt       dir1       dir2

$ cat  date.txt
Wednesday August 24 11:40:55 2016

$ cat   pwd.txt      ---> look at output of pwd.
/home/glass
$ _
```