UNIX Shells

• INTRODUCTION

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

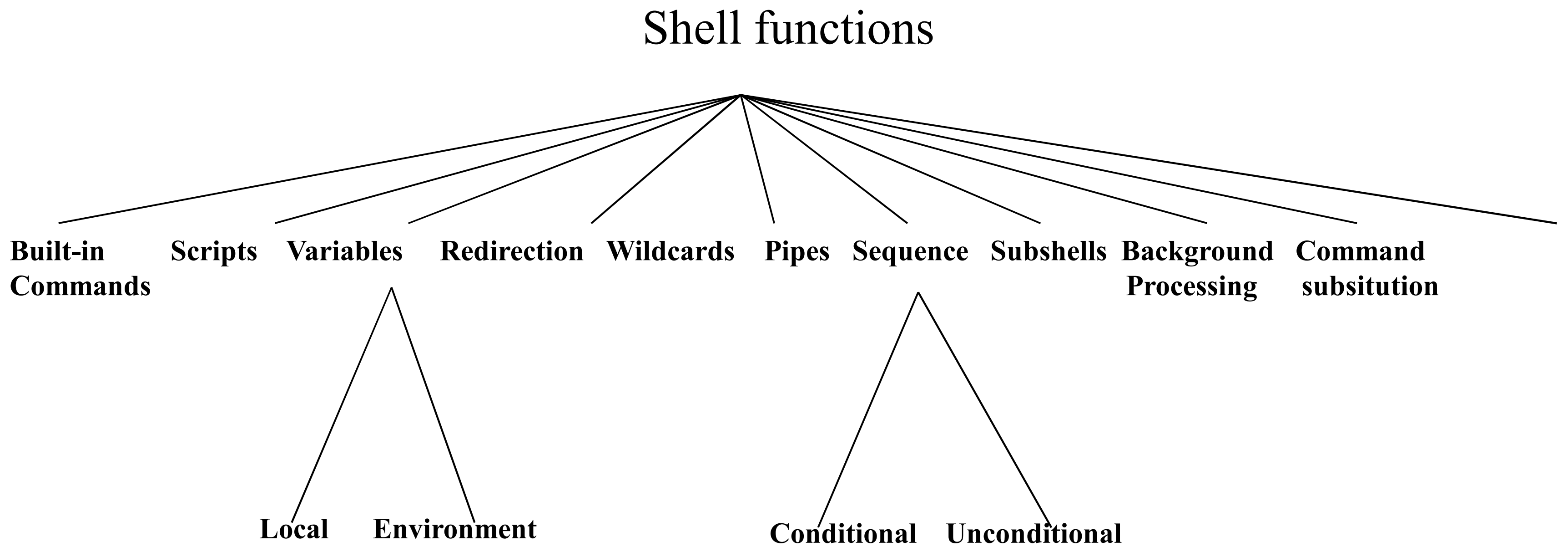
- Multitasking and piping
- Wildcards
- I/O redirection
- ETC.

There are four common shells in use:

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

• SHELL FUNCTIONALITY

-



• SHELL OPERATIONS

shell is invoked,
- either automatically during a login
- manually from a keyboard or script

1. 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.
$ _
```

METACHARACTERS

Some characters are processed specially

All four shells share a core set

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.

Symbol	Meaning
\`command\`	Command substitution; replaced by the output from command.
|	Pipe symbol; sends the output of one process to the input of another.
;	**Used to sequence commands.**
||	Conditional execution; executes a command if the previous one fails.
&&	Conditional execution; executes a command if the previous one succeeds.
(...)	Groups commands.
&	Runs a command in the background.
#	All characters that follow up to a new line are ignored by the shell and program(i.e., used for a comment)
$	Expands the value of a variable.
\	Prevents special interpretation of the next character.
<<tok	Input redirection; reads standard input from script up to tok.

To turn off 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.
$ cat file          ---> look at the file again.
hi
$ _
```

• Redirection

The shell redirection facility allows you 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

```
$ command > fileName
```

sends the standard output to fileName.

```
$  command >> fileName
```

appends the standard output to fileName.

• Input Redirection

Input redirection is useful because it allows you to prepare a process input beforehand and store it in a file for later use.

```
$ command < fileName
```

executes command using the file filename as its standard input.

- **FILENAME SUBSTITUTION(WILDCARDS)**

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.

```
$ ls -R     ---> 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 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
```

- **PIPES**

- uses the standard output of one process as the standard input of another process

- The sequence

```
$ command1 | command2
```

- 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 | wc -l
      5
```

```
$ who  |  tee who.capture  |  sort
ables      ttyp6    May   3   17:54  ( waterloo.com )
glass      ttyp0    May   3   18:49  ( bridge05.utdalla )
posey      ttyp2    May  23   17:44  ( blackfoot.utdall )
posey      ttyp4    May  23   17:44  ( blackfoot.utdall )

$ cat  who.capture    ---> look at the captured data.
glass      ttyp0    May   3   18:49  ( bridge05.utdalla )
posey      ttyp2    Apr  23   17:44  ( blackfoot.utdall )
posey      ttyp4    Apr  23   17:44  ( blackfoot.utdall )
ables      ttyp6    May   3   17:54  ( waterloo.com )
```

- **Utility : tee -ia   -{fileName}+**

The tee utility copies its standard input to the specified files and to its standard output.
The -a option causes the input to be appended to the files rather than overwriting them.
The -i option causes interrupts to be ignored.

COMMAND SUBSTITUTION

A command surrounded by back quote(`) is executed,

```
$ echo  the date today is `date`
the date today is Mon Feb 2 00:41:55 CST 1998

$ 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
$ _


$ vi  `grep  -l   debug  *.c`
```

- **SEQUENCES**

```
$ date;  pwd;   ls     ---> execute three commands in sequence.
Mon Feb  2  00:11:10  CST  1998
/home/glass/wild
a.c    b.c    cc.c       dir1      dir2
```

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

 $ cat  date.txt
 Mon  Feb  2  00:12:16   CST    1998

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

• GROUPING COMMANDS

- Commands may be grouped by placing them between parentheses,

- The group of commands shares the same standard input, standard output, and standard error channels

```
$ date; ls; pwd > out.txt                  ---> execute a sequence.
 Mon Feb 2 00:33:12  CST  1998   ---> output from date.
 a.c          b.c                          ---> output from ls.
 $ cat  out.txt                            ---> only pwd was redirected.
  /home/glass

$ ( date; ls; pwd ) > out.txt          ---> group and then redirect.
 $ cat out.txt                              ---> all output was redirected.
 Mon   Feb   2   00:33:28   CST   1998
 a.c                  b.c
 /home/glass
 $ _
```

- **Background Processing**

 - If you follow a simple command, pipeline, sequence of pipelines, or group of commands by the “&” metacharacter to execute the commands as a background process

 - does not take control of the keyboard.

```
$ find  .   -name    a.c    -print      --->search for “a.c”
./wild/a.c
./reverse/tmp/a.c

$ find  .   -name  b.c      -print &   --->search in the background.
27174                                   --->process ID number.
$ date                                  --->run “date” in the foreground.
./wild/b.c                              --->output from background “find”.
Mon  Feb  2  18:10:42  CST  1998 --->output from date.
$ ./reverse/tmp/b.c                     --->more output from background “find”
                                         --->came after we got the shell prompt,
```

• REDIRECTIONAL BACKGROUND PROCESSES

- Redirecting Ouput

```
$ find  .  -name  a.c  -print  > find.txt  &
27188                            ---> process ID of "find".

$ ls -l  find.txt                ---> look at "find.txt".
-rw-r--r--   1   glass    0  Feb   3   18:11   find.txt
$ ls -l  find.txt                ---> watch it grow.
-rw-r--r--   1   glass   29  Feb   3   18:11   find.txt

$ cat  find.txt                  ---> list "find.txt".
./wild/a.c
./reverse/tmp/a.c
$ _
```

• SHELL PROGRAMS: SCRIPTS

- Any series of shell commands in a regular text file for later execution.

- must give it execute permission by using the **chmod** utility.

to run it, you need only to type its name.

- Here are the rules that it uses to make this decision:

1) a pound sign(#) - the shell from which you executed

2) the form #! path name - *pathName* is the script.

3) neither rule1 nor rule2 - Bourne shell (sh).

• SHELL PROGRAMS: SCRIPTS

```
$ cat  >  script.sh                    ---> create  the bash script.
#! /bin/bash
# This is a sample bash script.
echo  -n  the date today is   # in bash,  -n omits new line
date  # output today’s date.
^D                                     ---> end of input.
$ chmod +x  script.sh                  ---> make the scripts executable.

$ ./script.sh                          ---> execute the shell script.
The date today is Sun Feb 1  19:50:00  CST  2004
```

• VARIABLES

- Two kinds of variables:
  - local variables
  - environment variables

predefined environment variables

Name	Meaning
$HOME	the full pathname of your home directory
$PATH	a list of directories to search for commands
$MAIL	the full pathname of your mailbox
$USER	your username
$SHELL	the full pathname of your login shell
$TERM	the type of your terminal

• VARIABLES

- To acess variables : a variable with a $
  - $HOME

- To create a variable : simply assign it a value
  - variable does not have to be declared.

- assigning a variable

  variableName=value ---> place no spaces

  or

  variableName=" value " ---> spacing doesn't matter.

```
$ echo HOME = $HOME, PATH=$PATH     ---> list two variables.
HOME =/home/glass, PATH=/bin:/usr/bin:/usr/sbin

$ echo USER = $USER, SHELL = $SHELL, TERM=$TERM
USER = glass, SHELL = /bin/sh, TERM=vt100
```

- several common built-in variables that have special meanings:

Name	Meaning
$$	The process ID of the shell.
$0	The name of the shell script(if applicable).
$1..$9	$n refers to the nth command line argument (if applicable).
$*	A list of all the command-line arguments.
$?	Exit status of last command. Set by the exit command

```
$ cat  script.sh          ---> list the script.
echo the name of this script is $0
echo the first argument is $1
echo a list of all the arguments is $*
echo this script places the date into a temporary file
echo called $1.$$
date > $1.$$     # redirect the output of date.
ls    $1.$$         # list the file.
rm    $1.$$         # remove the file.
```

```
$ script.sh  paul  ringo  george  john   ---> execute the script.
the name of this script is script.sh
the first argument is paul
a list of all the arguments is paul ringo george john
this script places the date into a temporary file
called paul.24321
paul.24321
```

• QUOTING

- to inhibit the shell’s wildcard-replacement, variable-substitution, and/or command-substitution mechanisms.

1. Single quotes(‘) inhibit wildcard replacement, variable substitution, and command substitution.
2. Double quotes(“) inhibit wildcard replacement only.
3. When quotes are nested, it’s only the outer quotes that have any effect.

```
$ echo  3 * 4 = 12    ---> remember, * is a wildcard.
3   a.c    b    b.c    c.c    4 = 12
$ echo  “3 * 4 = 12” ---> double quotes inhibit wildcards.
3 * 4 = 12
$ echo  ‘3 * 4 = 12’  ---> single quotes inhibit wildcards.
3 * 4 = 12

$ name=Graham
$ echo  ‘my name is $name - date is `date`’
my name is $name - date is ‘date’

$ echo  “my name is $name - date is `date`”
my name is Graham - date is Mon Feb 2 23:14:56 CST 1998
```

- **JOB CONTROL**

There are two utilities and one built-in command that allow you to do so:

1) **ps**, which generates a list of processes and their attributes, including their names, process ID numbers, controlling terminals and owner.

2) **kill**, which allows you to terminate a process based on its ID number.

3) **wait**, which allows a shell to wait for one of its child processes to terminate.

- **Process Status: ps**

Utility : ps -efl

a listing of process-status information.

-e : all running processes.
-f : full listing.
-l : long listing.

Col	Meaning
S	the state of the process
UID	the effective user ID of the process
PID	the ID of the process
PPID	the ID of the parent process
C	the percentage of CPU time
PRI	the priority of the process
SZ	the size of the process’ data in kilobytes
STIME	the time the process was created
TTY	the controlling terminal
TIME	the amount of CPU time used so far
CMD	the name of the command

```
$ ( sleep 10; echo done ) &     ---> delayed echo in background.
27387                            ---> the process ID number.
$ ps
PID     TTY     TIME     CMD
27355  pts/3  0:00  -sh
27387  pts/3  0:00  -sh
27388  pts/3  0:00  sleep 10       ---> the sleep.
27389  pts/3  0:00  ps            ---> the ps command itself!
$ done                           ---> the output from the background
                                      process.
```

Utility : sleep seconds

The sleep utility sleeps for the specified number of seconds and then terminates.

• Signaling Processes:kill

- to terminate a process before it completes, use the ***kill*** command.

Utility: kill [-signalId] {pid}+
kill -l

- kill sends the signalId to the list of numbered processes.
- signalId may be the number or name of a signal.
- To obtain a list of the legal signal names, use the -l option.

- To kill a process, you must either own it or be a super-user.

```
$ ( sleep 10; echo done ) &
 27390                                    ---> process ID number.
 $ kill -9 27390                          ---> kill the process.
 $ ps                                     ---> it's gone!
 PID    TT    STAT    TIME    COMMAND
 27355     p3    S            0:00   -sh(sh)
 27394     p3    R            0:00   ps
```

```
$ sleep 30 & sleep 30 & sleep 30 & ---> create three processes.
27429
27430
27431

$ kill  0                                      ---> kill them all.
27431  Terminated
27430  Terminated
27429  Terminated

$ _
```

- **FINDING A COMMAND: $PATH**

When a shell processes a command,
it first checks to see whether it’s a built-in command;

- echo is an example of a built-in shell command

- If the file doesn’t exist or isn’t an executable, an errors occurs:

```
$ /bin/ls                              ---> full pathname of the ls utility.
script.csh  script.ksh
$ /bin/nsx                             ---> a nonexistent filename.
/bin/nsx: not found
$ /etc/passwd                          ---> the name of the password file.
/etc/passwd: Permission denied  ---> it’s not executable.
$ _
```

- **OVERLOADING STANDARD UTILITIES**

```
$ mkdir  bin            ---> make my own personal “bin” directory.
$ cd bin                ---> move into the new directory.
$ cat > ls              ---> create a script called “ls”.
echo my ls
^D                      ---> end of input.
$ chmod +x  ls          ---> make it executable.

$ echo $PATH            ---> look at the current PATH setting.
/bin:/usr/bin:/usr/sbin
$ echo $HOME            ---> get pathname of my home directory.
/home/glass
$ PATH=/home/glass/bin:$PATH    ---> update.
$ ls                    --->  call “ls”.
my ls                   ---> my own version overrides “/bin/ls”.
$ _
```

.

- **SUBSHELLS**

When you log into a UNIX system, you execute an initial login shell.

This initial shell executes any simple commands that you enter.

- current(parent) shell creates a new(child) shell to perform some tasks:

1) When a grouped command,
such as (ls; pwd; date), is executed, the parent shell creates a child shell to execute the grouped commands.

If the command in not executed in the background,
the parent shell sleeps until the child shell terminates.

2) When a script is executed,
the parent shell creates a child shell to execute the commands in the script.

If the script is not executed in the background,
the parent shell sleeps until the child shell terminates.

3) When a background job is executed,
The parent shell creates a child shell to execute the background commands.

The parent shell continues to run concurrently with the child shell.