Chapter 4: The UNIX Shells (Bourne shell, Korn shell, C shell)

Graham Glass and King Ables,
UNIX for Programmers and Users,

Original Notes by Raj Sunderraman
Converted to Presentation and Updated by Bing Li
The Relationship of Shell Functionality
Which Shell

- To change your default shell use `chsh` utility
  - `$ chsh`
    - modify profile
- To examine your default shell, type:
  - `$ echo $SHELL`
    - output environment variable `$SHELL`
Core Shell Functionality

- Built-in commands
- Scripts
- Variables (local, environment)
- Redirection
- Wildcards
Core Shell Functionality

- Pipes
- Sequences (conditional, unconditional)
- Subshells
- Background processing
- Command substitution
Invoking the Shell

• A shell is invoked, either
  • automatically upon login, or
  • manually from the keyboard or script
What does the shell do?

- The following takes place:

1. reads a special startup file (.bashrc or bash in the user’s home directory) and executes all the commands in that file.

2. displays a prompt and waits for a user command

3. If user enters CTRL-D (end of input) the shell terminate, otherwise it executes the user command(s)
User Commands

• e.g.: ls, cat, chmod…

• Programs in /bin/ or other place.
Built-in commands

• Most UNIX commands invoke utility programs stored in the file hierarchy

• The shell has to locate the utility (using PATH variable)

• Shells have built-in commands
  • e.g.: cd, exit, man
Metacharacters

• Output redirection
  • > : writes standard output to file
  • >> : appends standard output to file

• Input redirection
  • < : reads standard input from file
  • << tok : reads standard input until meets a line containing tok
Metacharacters

• File-substitution wildcards:
  • * matches 0 or more characters
  • ? matches any single character
  • […] matches any character within brackets

• Command substitution:
  • `command` replaced by the output of command
  • e.g.: echo `ls`
Metacharacters

• |   pipe

• send output of one process to the input of another
  • e.g.: list files, then use word count to count lines
    • ls | wc -l
  • this effectively counts the files
Metacharacters

• ;    used to sequence commands

• Conditional execution
  • ||    execute command if previous one fails
  • &&    execute command if previous one succeeds
Metacharacters

• (...) Group commands
• & Run command in background
• # Comment
  • rest of characters ignored by shell
• $ Expand the value of a variable
• \ Prevent special interpretation of character that follows
Redirection

• The shell redirection facility allows you to
  • store the output of a process to a file
  • use the contents of a file as input to a process

• Examples:
  • cat x1.c > y.c
  • cat x2.c >> y.c
  • mail tony < hiMom

• The << tok redirection is almost exclusively used in shell scripts
Filename Substitution

- $ ls *.c  # list .c files
- $ ls ?.c  # list files like a.c, b.c, 1.c, etc.
- $ ls [ac]*  # list files starting with a or c
- $ ls [A-Za-z]*  # list files beginning with a letter
- $ ls dir*//*.c  # list all.c files in directories starting with dir
Pipes

- $ command1 | command2 | command3
- $ ls | wc -w
  - 4
Command Substitution

• A command surrounded by grave accents (``) is executed and its standard output is inserted in the command’s place in the command line.

• $ echo today is `date`
  
  • today is Sun February 14 18:53:08 EDT 2016

• echo there are `who | wc -l` users on the system
  
  • There are 2 users on the system
Sequences

• Commands or pipelines separated by semicolons

• Each command in a sequence may by individually I/O redirected

• Example

  • $ date; pwd; ls

  • $ date > date.txt; pwd > pwd.txt; ls
Sequences

• Conditional sequences:
  
  • $ cc myprog.c && a.out
  
  • cc mypgrog.c || echo compilation failed

• In a series of commands separated by &&, the next command is executed if the previous one succeeds (returns an exit code of 0)

• In a series of commands separated by || the next command is executed if the previous one fails (return an exit code of non-zero)
Grouping Commands

• Commands can be grouped by putting them within parentheses
  • a sub shell is created to execute the grouped commands
• Example:
  • $(date; ls; pwd) > out.txt
  • $ more out.txt
Background Processing

- An & sign at the end of a simple command
  - or pipeline, sequence of pipelines
  - or a group of commands
- Starts a sub-shell
  - Commands are executed as a background process
  - Does not take control of the keyboard
- A process ID is displayed when it begins
Background Processing

- Redirect the output to a file (if desired)
  - prevents background output on terminal
- Background process cannot read from standard input
  - if they attempt to read from standard input; they terminate.
Shell Programs/Scripts

- Shell commands may be stored in a text file for execution.
- Use the `chmod` utility to set execute permissions on the file
  - e.g.: `chmod +x script_file`
- Executing it by simply typing the file name (`./script_file_name`)
- When a script runs, the system determines which shell to use
Shell Programs/Scripts

• To determine which shell:

  • if the first line of the script is a pound sign (#), then the script is interpreted by the current shell

  • if the first line of script is one of the form (#!/bin/sh or #!/bin/ksh etc.), then the appropriate shell is used to interpret the script

  • else, the script is interpreted by the Bourne shell

• Note: pound sign on 1st column in any other line implies a comment line
Shell Programs/Scripts

• Always recommended to use `#!/pathname`

```
#!/bin/csh
# A simple C-shell script
echo -n "The date today is"
date
```
Subshells

• Several ways a subshell can be created:
  • Grouped command (ls;pwd;date)
  • Script execution
  • Background processes
• A subshell has its own working directory
  • cd commands in subshells do not change working directory of parent shell
Subshells

- Every shell has two data areas
  - environment space
  - local-variable space
- Child shell gets a copy of the parent’s environment space
  - starts with an empty local-variable space.
Variables

• A shell supports two kinds of variables:
  • Local variables
  • Environment variables
  • Both hold data in string format
• Every shell has a set of pre-defined environment variables and local variables.
• Accessing variables in all shells is done by prefixing the name with a $ sign
Variables

• Some pre-defined environment variables available in all shells:
  • $HOME, $PATH, $MAIL, $USER, $SHELL, $TERM
Assigning Values to Variable

- Depends on shell
  - sh, bash, ksh: variable=value , variable=“value”
  - Notice no spaces around equal sign
- To make a variable an environment variable in sh,bash,ksh
  - export variable
Assigning Values to Variables

- csh: `set variable=value`, `set variable="value"

- To assign environment variables
  - `setenv TERM vt100`
Built-in Variables

• Common built-in variables with special meaning:
  • $$  process ID of shell
  • $0  name of shell script (if applicable)
  • $1..$9  $n refers to the $n$th command line argument (if applicable)
  • $*  a list of all command line arguments
Example Using Built-in Variable

$ cat script2.csh
#!/bin/csh
echo the name of this file is $0
echo the first argument is $1
echo the list of all argument is $*
echo this script places the date into a temporary file called $1.$$
date > $1.$$
ls -l $1.$$ 
rm $1.$$
Running the Example

$ ./script2.csh paul ringo george john
the name of this file is ./script2.csh
the first argument is paul
the list of all arguments is paul ringo george john
this script places the date into a temporary file called paul.554
- r w - r w - r - - 1   bing   bing  29 Jun 20 21:33 paul.554
Quoting

- Single quotes (‘) inhibit wildcard replacement, variable substitution, and command substitution
- Double quotes (“) inhibit wildcard replacement only
- When quotas are nested only the outer quotes have any effect
Quoting Examples

$ echo 3 * 4 = 12
3 Downloads Desktop ... 4 = 12

$ echo '3 * 4 = 12'
3 * 4 = 12

$ echo “my name is $USER; the date is `date`”
my name is bing; the date is Tue Feb 23 23:08:10 EST 2016
Here Documents

$ cat here.csh
mail $1 <<< ENDOFTEXT
Dear $1,
Please see me regarding some exciting news!
$USER
ENDOFTEXT
echo mail sent to $1

$ ./here.csh bing
mail sent to bing
Job Control

• `ps` command generates a list of processes and their attributes

• `kill` command terminates processes based on process ID

• `wait` allows the shell to wait for one of its child processes to terminate.
ps Command

• $ ps -efl
  • e: include all running processes
  • f: include full listing
  • l: include long listing
# ps Command

<table>
<thead>
<tr>
<th>Column</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>S</td>
<td>the process state</td>
</tr>
<tr>
<td>UID</td>
<td>The effective user ID of the process</td>
</tr>
<tr>
<td>PID</td>
<td>The process ID</td>
</tr>
<tr>
<td>PPID</td>
<td>The parent process ID</td>
</tr>
<tr>
<td>C</td>
<td>The percentage of CPU time that the process used in the last minute</td>
</tr>
<tr>
<td>PRI</td>
<td>The priority of the process</td>
</tr>
<tr>
<td>SZ</td>
<td>The size of the process's data and stack in kb</td>
</tr>
<tr>
<td>STIME</td>
<td>The time the process was created</td>
</tr>
<tr>
<td>TTY</td>
<td>The controlling terminal</td>
</tr>
<tr>
<td>TIME</td>
<td>The amount of CPU time used so far (MM:SS)</td>
</tr>
<tr>
<td>CMD</td>
<td>The name of the command</td>
</tr>
</tbody>
</table>
# ps Command

<table>
<thead>
<tr>
<th>Letter</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>O</td>
<td>Running an a processor</td>
</tr>
<tr>
<td>R</td>
<td>Runnable</td>
</tr>
<tr>
<td>S</td>
<td>Sleeping</td>
</tr>
<tr>
<td>T</td>
<td>Suspended</td>
</tr>
<tr>
<td>Z</td>
<td>Zombie process</td>
</tr>
</tbody>
</table>
nohup Command

- Bourne and Ksh automatically terminate background processes when you log out (csh allows them to continue)

- To keep the background processes to continue in sh and csh, use
  - `$ nohup command`

- Use `ps -x` to see
# Signaling Processes: `kill`

<table>
<thead>
<tr>
<th>Signaling Process</th>
<th><code>$ kill -l</code> Command</th>
</tr>
</thead>
<tbody>
<tr>
<td>HUP</td>
<td>INT</td>
</tr>
<tr>
<td>ILL</td>
<td>TRAP</td>
</tr>
<tr>
<td>EMT</td>
<td>FPE</td>
</tr>
<tr>
<td>SEGV</td>
<td>SYS</td>
</tr>
<tr>
<td>ALRM</td>
<td>TERM</td>
</tr>
<tr>
<td>STOP</td>
<td>TSTP</td>
</tr>
<tr>
<td>CHLD</td>
<td>TTIN</td>
</tr>
<tr>
<td>IO</td>
<td>XCPU</td>
</tr>
<tr>
<td>VTALRM</td>
<td>PROF</td>
</tr>
<tr>
<td>INFO</td>
<td>USR1</td>
</tr>
<tr>
<td>BUS</td>
<td>USR2</td>
</tr>
</tbody>
</table>

The signal commands allow for the process to be terminated or signaled with specific actions. For example, `$ kill -l` lists all available signals. Each signal type corresponds to a specific action when sent to a process.
kill

- $ kill -signal pid
- if signal is not specified the default signal is TERM
- KILL is useful if the process refuse to die
- $(sleep 20; echo done) &
- $kill
Waiting for Child Processes

• A shell may wait for one or more of its child processes to terminate by using a built-in command: `wait [pid]`

```
$ (sleep 30;echo done 1) &
[1] 429
$ (sleep 30;echo done 2) &
[2] 431
$ echo done 3; wait; echo done 4
done 3
done 1
[1] - 429 done (sleep 30; echo done 1)
done 2
[2] + 431 done (sleep 30; echo done 2)
done 4
```
Finding a Command: $PATH

- if the command is a shell built-in such as `man` or `cd` it is directly interpreted by the shell.
- if the command begins with a `/`
  - shell assumes that the command is the absolute path name of an executable
  - error occurs if the executable is not found.
- if not built-in and not a full pathname
  - shell searches the directories in the PATH
  - from left to right for the executable
- current working directory may not be in PATH
PATH variable

• if PATH is empty or is not set, only the current working directory is searched for the executable.

• Home-brewed utilities:
  • Some UNIX users create their own utilities
  • Stored in their bin directory
  • Place their bin directory ahead of all others
  • Their version of the utility is executed
  • PATH=/homebrewed/bin:$PATH
Termination and Exit codes

- Every UNIX process terminates with an exit value.
- By convention, a 0 value means success and a non-zero value means failure.
- All built-in commands return 1 when they fail.
Termination and Exit Codes

• The special variable `$?` contains the exit code of the last command execution. In csh `$status` also contains the exit code.

• Any script written by you should contain the exit command:
  
  • exit <number>

• if the script does not exit with a exit code, the exit code of the last command is returned by default.
Common Core Built-in Commands

• `eval` command

  • The eval shell command executes the output of the command as a regular shell command

  $eval `echo x=5`
  $echo $x
  5
Common Core Built-in Commands

- **exec** command

  - The exec shell command causes the shell’s image to be replaced with the command in the process’ memory space.

  - As a result, if the command terminates, the shell also ceases to exist; if the shell was a login shell, the login session terminates.
Common Core Built-in Commands

- **shift**

  - This command cause all of the positional parameters $2..n$ to be renamed $1..(n-1)$ and $1$ is lost.

  - Useful in processing command line parameters.
Common Core Built-in Commands

$ cat script3.csh
#!/bin/csh
echo first argument is $1, all args are $*
shift
echo first argument is $1, all args are $*

$./script3.csh a b c d
first argument is a, all args are a b c d
first argument is b, all args are b c d
umask Command

• Every UNIX process has a special quantity called umask value
  • default value: 022 octal
• Whenever a file is created
  • e.g. made by vi or by redirection
  • File permissions (usually 666) masked (XORed) with umask value
  • Example to produce the permission 644
umask Command

- To see current umask value
  - $umask
- To change umask value
  - $umask octalValue
Review

- Covered core shell functionality
  - Built-in commands
  - Scripts
  - Variables
  - Redirection
  - Wildcards
  - Pipes
  - Subshells
  - Background processing