Ch. 3 The UNIX Shells (Bourne shell, Korn shell, C shell)

- To change your default shell use the chsh utility
- To examine your default shell, type

```
echo $SHELL
```

CORE Shell Functionality:

- Built-in commands
- Scripts
- Variables (local, environment)
- Redirection
- Wildcards
- Pipes
- Sequences (conditional, unconditional)
- Subshells
- Background processing
- Command substitution

What does the shell do?

When a shell is invoked, either automatically upon login or manually from the keyboard or script, the following takes place:

(1) It reads a special startup file (.cshrc for csh in the user's home directory) and executes all the commands in that file
(2) It displays a prompt and waits for a user command
(3) If user enters CTRL-D (end of input); the shell terminates otherwise it executes the user command

User commands:

```
$ ls

$ ps -ef | sort | ul -tdumb | lp

$ ls | sort | \
  lp
```

Built-in commands

- Most Unix commands invoke utility programs stored in the file hierarchy (ex. ls, vi etc); The shell has to locate the utility in the file system (using PATH variable)

- Shells have built-in commands; Two important ones: echo, cd

- echo arguments

```
$ echo Hi, How are you?
Hi, How are you?
```

  echo by default appends a new line (to inhibit new line use -n optio
  in csh)

- cd dir

Metacharacters

`>`	Output redirection (writes std. output to file)
`>>`	Output redirection (appends std. output to file)
`<`	Input redirection (reads std. input from file)
`*`	File-substitution wildcard; matches 0 or more characters
`?`	File-substitution wildcard; matches any single character
`[...]`	File-substitution wildcard; matches any character within brackets
‘command‘	Command substitution; replaced by the output of command
`|`	Pipe; send output of one process to the input of another
`;`	Used to sequence commands
`||`	Conditional execution; execute command if previous one fails
`&&`	Conditional execution; execute command if previous one succeeds
`(...)`	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
`<<tok`	Input redirection; read std. input until tok.

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

- cat x1.c > y.c
- cat x2.c >> y.c
- mail tony < hiMom

- The <<tok redirection is almost exclusively used in shell scripts
   (will see this later)
```

Filename substitution

```
$ ls *.c        # list .c files
$ ls ?.c        # list files such as 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
ppp00*  ppp24*  ppp48*  ppp72*
$ ls | wc -w
      4
$ head -4 /etc/passwd
root:fjQyH/FG3TJcg:0:0:root:/root:/bin/sh
bin:*:1:1:bin:/bin:
daemon:*:2:2:daemon:/sbin:
adm:*:3:4:adm:/var/adm:

$ cat /etc/passwd | awk -F: '{print $1}' | sort
adm
bin
daemon
raj
```

tee utility

```
$ tee -ia filename
```

causes standard input to be copied to file and also sent to standard output. (-a option appends to file; -i option ignores interrupts)

```
$ who
raj      tty1     Jun 19 09:31
naveen   ttyp0    Jun 19 20:17 (localhost)

$ who | tee who.capture | sort
naveen   ttyp0    Jun 19 20:17 (localhost)
raj      tty1     Jun 19 09:31

$ more who.capture
raj      tty1     Jun 19 09:31
naveen   ttyp0    Jun 19 20:17 (localhost)
```

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 Sat Jun 19 22:23:28 EDT 1999


$ echo there are ‘who | wc -l‘ users on the system
there are 2 users on the system
```

Sequences

- Commands or pipelines separated by semi-colons
- Each command in a sequence may be individually I/O redirected.

```
$ date; pwd; ls
Sat Jun 19 22:33:19 EDT 1999
/home/raj/oracle
jdbc/   ows/    proc/   sql/    sqlj/   who.capture
$ date > date.txt; ls pwd > pwd.txt
```

- Conditional sequences:

```
$ cc myprog.c && a.out
$ cc myprog.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 (returns 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)

```
$ (date; ls; pwd) > out.txt
$ more out.txt
Sat Jun 19 22:40:43 EDT 1999
date.txt
jdbc/
out.txt
ows/
proc/
pwd.txt
sql/
sqlj/
who.capture
/home/raj/oracle
```

Background processing

- An & sign can follow a simple command, pipeline, sequence of pipelines, or a group of commands

- This starts a sub-shell and the commands are executed from the sub-shell as a background process which does not take control of the keyboard.

- A process id is displayed when a background process begins

- To prevent output from a background process to come to the terminal, you may redirect the output to a file.

- Background process cannot read from standard input; If they attempt to read from standard input; they terminate.

Shell Programs/Scripts

- Any series of shell commands may be stored in a text file for execution.

- Use the chmod utility to set execute permissions on the file before executing it by simply typing the file name.

- When a script runs, the system determines which shell the script was written for; The rules are:
  * if the first line of the script is a pound sign (#), then the script is interpreted by the shell from which the script is executed.
  * if the first line of the script is 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

- Always recommended to use #!pathname

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

Subshells

- Within a login shell there are 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 subshell do not change working directory of parent shell

- Every shell has two data areas: an environment space and a local-variable space; When a child shell is created it gets a copy of the parent’s environment space but starts with an empty local-variable space.

Variables

- A shell supports two kinds of variables: local and environment variables. Both kinds hold data in string format.

- Every shell has a set of pre-defined environment variables and local variables. Some pre-defined environment variables available in all shells:

  $HOME, $PATH, $MAIL, $USER, $SHELL, $TERM

- Accessing variables in all shells is done by prefixing the name with a $ sign.

- Assigning values to variables is done differently in different shells:

```
sh, ksh:  variable=value
          variable = "value"
     To make a variable an environment variable in sh, ksh
          export variable

csh:      set variable=value
          set variable = "value"
     To assign environment variables
     setenv TERM vt100
```

- Common built-in variables with special meaning:

`$$`	process ID of shell
`$0`	name of shell script (if applicable)
`$1..$9`	`$n` refers to the nth command line argument (if applicable)
`$*`	a list of all command line arguments

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

$ 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
-rw-rw-r--   1 raj      raj            29 Jun 20 21:33 paul.554
```

Quoting

- Single quotes (') inhibit wildcard replacement, variable substitution, and command substitution

- Double quotes (") inhibits wildcard replacement only

- When quotes are nested only the outer quotes have any effect

```
$ echo 3 * 4 = 12
3 3.log 3.tex script.csh script2.csh 4 = 12

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

$ echo "my name is $USER; the date is `date`"
my name is raj; the date is Sun Jun 20 21:59:13 EDT 1999
```

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 raj
mail sent to raj
```

```
$ mail
Mail version 8.1 6/6/93.  Type ? for help.
"/var/spool/mail/raj": 6 messages 1 new
    5 raj@kamakshi.gsu.edu  Sun Jun 20 22:13  18/420
>N  6 raj@kamakshi.gsu.edu  Sun Jun 20 22:14  14/377
&
Message 6:
From raj  Sun Jun 20 22:14:31 1999
Date: Sun, 20 Jun 1999 22:14:31 -0400
From: raj@kamakshi.gsu.edu
To: raj@kamakshi.gsu.edu

Dear raj,
Please see me regarding some exciting news!
raj
```

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 -efl # e: include all running processes
          # f: include full listing
          # l: include long listing

PID : process ID

- 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 ksh, use

$ nohup command
```

Signalling processes: kill

```
$ kill -l
 1) SIGHUP       2) SIGINT       3) SIGQUIT      4) SIGILL
 5) SIGTRAP      6) SIGIOT       7) SIGBUS       8) SIGFPE
 9) SIGKILL     10) SIGUSR1     11) SIGSEGV     12) SIGUSR2
13) SIGPIPE     14) SIGALRM     15) SIGTERM     17) SIGCHLD
18) SIGCONT     19) SIGSTOP     20) SIGTSTP     21) SIGTTIN
22) SIGTTOU     23) SIGURG      24) SIGXCPU     25) SIGXFSZ
26) SIGVTALRM   27) SIGPROF     28) SIGWINCH    29) SIGIO
30) SIGPWR

$ kill -signal pid
if signal is not specified the default signal is SIGTERM (15)
SIGKILL (9) is useful if the process refuses to die.
```

Waiting for child processes

```
$ (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]-  Done                    ( sleep 30; echo done 1 )
done 2
[2]+  Done                    ( sleep 30; echo done 2 )
done 4
```

This feature is used in advanced shell scripts.

Finding a command: $PATH

- If the command is a shell built-in such as echo or cd it is directly interpreted by the shell.

- if the command begins with a / the shell assumes that the command is the absolute path name of an executable; error occurs if the executable is not found.

- if the command is not a built-in and not a full pathname, the shell searches the directory names that are stored in the PATH environment variable from left to right for an executable that matches the command.

- Normally, the current working directory is included in the PATH variable.

- If PATH is empty or is not set, only the current working directory is searched for the executable.
- Homebrewed utilities: Some Unix users create their own versions of some Unix utilities and store them in their bin directory; Then they place their bin directory ahead of all other directories so that their version of the utility is executed.

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

- 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
  ```

- 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.

- shift

  This command causes all of the positional parameters $2..$n
  to be renamed $1..$(n-1) and $1 is lost.
  Useful in processing command line parameters.

```
#!/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
```

- Every Unix process has a special quantity called umask value. The default value is 022 octal
- Whenever a file is created (say by vi or by redirection), the file permissions, which is usually 666, is masked (xor) with umask value say 022 to produce the permission 644
- To change umask value use the command

```
$ umask octalValue
```

- To see current umask value use the command

```
$ umask
```