CHAPTER II: UNIX SHELLS

B. Boufama

University of Windsor

Introduction

A shell is a program that acts as an interface between a user and the operating system.
The shell starts running as soon as you log on.
A Unix shell is basically a *command interpreter*.
A shell command can be

- internal(built-in) : the code to be executed is part of the shell or,
- external : the code to be executed resides in a sperate binary file.

Because the shell accepts command from the keyboard, it terminates when the special character CTR-D is entered.

Path and external commands

For an external command, the shell searches for its file in the directories whose names are stored in the shell variable **PATH**.

How to define the shell varibale **PATH**?

Example: $PATH = ./ : /usr/bin :\sim /bin$

$\rightarrow$ the shell looks for the command in order in

- the current directory, called **./**,
- /usr/bin/
- your-login-directory/bin/

You can also add more directories to your current path by :

PATH=/usr/local/bin:$PATH
You can view your path : echo $PATH

The command *which command-name* allows you to find the file location of the command.
Examples: which echo, which emacs

Note that PATH is set in a shell start-up file,
Example : in the file ∼/.profile,
where ∼/ represents the current user loggin directory.

Shells

Popular Unix shells are :

- Bourne Again shell : **bash** widely used, superset of Bourne shell **sh**.
- C shell : **csh** richer than **sh**, syntax $\sim$ to C.
- Korn shell : **ksh** derived from **sh** with more functions.

You can get your running shell by typing

echo $SHELL

→ Running a shell

When a shell starts, it

1. reads special startup files (e.g., /etc/profile then ∼/.profile for bash) that contain some initialization information,
2. displays a prompt and waits for user commands then,
3. the shell executes the user's command and returns to step 2 unless the user has typed the characters **CTR-D** which will cause the shell to terminate.

→ Metacharacters

These are special characters with special meanings:

> : Output redirection
> E.g., ls > fileNames.txt

< : Input redirection
E.g., mailx user@uwindsor.ca < letter.txt

>> : Output redirection, appends to a file
E.g., ls >> fileNames.txt

***** : filename Wild card, matches 0 or more characters
E.g., rm *ps, delete all files ending with 'ps'.

? : filename Wild card, matches 1 character.
E.g., rm *.? → delete files with one character after '.'
ls ?? → list files/directories made up of 2 characters.

‘*command*‘ : command substitution, replaced by the command output.
E.g. 1. echo The date of today is ‘date‘
E.g. 2. echo hello ls → hello ls
echo hello ‘ls‘ → hello followed by the ls outputs.

| : Pipe between two commands.
E.g., ls | wc -w → output of ls is piped to wc to get the number of files/directories.
Note the utility **wc** displays a count of lines, words and characters in a file.

; : Used to sequence commands
E.g., date ; ls; date

& : Executes a command in the background
E.g., netscape &

: characters after this are ignored(comment)

$: Expands the value of a shell variable
E.g., echo $PATH

\ : Prevents special interpretation of next character.
E.g., echo this is \& → this is &
& is not a metacharacter in this case.

Shell programs: Scripts

A **shell script** is a text file containing shell commands.
A shell language has

- to define, read and write **shell variables**
- control structures such as loop and if statements

example:

```
#!/bin/bash

echo Today is ‘date +"%B %d, %Y"‘

ls $1/b???? | wc -w

ls $1/$2???? | wc -w
```

→ How to run a script ?

- Make the file executable :
  chmod +x *script_file*
  In this case, the *script_file* becomes like a command.
- By running the /bin/bash :
  bash *script_file*

→ Which shell to run for a script ?

Different shells have different syntax.
→ A C-shell script won't be run by bash shell.
The shell to be used for a script is chosen as follow :

- If the first line of the script consists of the only character # then the script is interpreted by the shell from which it has been called.
- If the first line of the script is of the form #!*fullPathName*, then the program *fullPathName* is used.
- Otherwise, the Bourne shell is used.

Shell variables

Two kinds of variables are supported by a shell, user-defined and shell environment variables.
Both variables store data as strings.

→ **Shell environment variables**
To customize your environment.
Most of these variables are initialized by the start-up file /etc/profile.
You can customize them by assigning them different values in your ∼/.profile file.

→ **User-defined variables**
Used within shell scripts as temporary storage.

You can use the **set** command to display all your shell variables.

→ Some important read/write shell environment variables

HOME	Full path name of your home directory
PATH	List of directories to search for commands
USER	Your user-name
SHELL	Your login shell
PWD	Current working directory

→ read-only shell variables

$0	name of the program that is runing
$1...$9	values of command line arguments 1 through 9
$⋆	values of all command line arguments
$#	total number of command line arguments
$$	Process ID of current process
$?	Exit status of most recent command
$!	PID of most recent background process

Quoting

The shell's wildcard/variable/command substitution mechanism can be inhibited using quotes:

- Single quotes (') inhibit wildcard/variable/command substitution.
- Double quotes (") inhibit wildcard replacement only.
- When quotes are nested, only the outer quotes mater.

Examples

echo 3 * 5=15 won't work because * is a wildcard
echo '3 * 5=15' → 3*5=15
echo 'I am $USER' → I am $USER
echo "I am $USER" → I am admin256

Case of bash as a programming language

In addition to the basic facilities, shells have buit-in programming languages that support in particular:

- conditions,
- loops,
- input/output
- basic arithmetics

Example 1

```
#!/bin/bash
echo -n "Enter a value> "
read a
echo -n "Enter another value> "
read b
echo "Doing arithmetics> "
# When assigning variables, no space on either
# side of the = sign. To do arithmetic in bash,
# surround expressions with $(( and ))$.
sum=$(( $a + $b ))
echo "The sum $a + $b is $sum"
difference=$(( $a - $b ))
echo "The difference $a - $b is $difference"
product=$(($a * $b))
echo "The product $a * $b is $product"
if [[ $b -ne 0 ]]; then
 quotient=$(($a / $b))
 echo "The division $a / $b is $quotient"
else
 echo "The division $a/$b is impossible"
fi
```

Example 1, enhanced

```
#!/bin/bash
if [ $# != 2 ]; then   # or, if ( test $# != 2 )
  echo "Usage: $0 integer1 integer2"
else
  echo Doing arithmetic>
  r=$(($1 + $2))
  echo "the sum $1 + $2 is $r"
  r=$(($1 - $2))
  echo "the subtraction $1 - $2 is $r"
  r=$(($1 * $2))
  echo "the product $1 * $2 is $r"
  if [ $2 -ne 0 ] ; then
    r=$(($1 / $2))
    echo "the division $1 / $2 is $r"
  else
    echo "the division $1 / $2 is impossible"
  fi
fi
```

- **Accessing a simple variable**:
  → $name: access value of variable name
  Example: dir=/export/home/
  echo "my home directory is ${dir}boufama/"
  → /export/home/boufama/

- **List variables**: **name=(arg1 arg2 ...)**
  names=(Windsor Toronto Ottawa)
  echo ${names[1]} → Toronto

  echo ${names[@]:1:2} → Toronto Ottawa
  (Two elements starting at index 1)

  echo ${names[*]} → Windsor Toronto Ottawa

  echo ${#names[@]} → 3 #number of elements

  names=(${names[@]} London) #add a new element
  names[1]=Quebec #change element 1 (2nd element)
  echo $names[@] → Windsor Quebec Ottawa London

- **String expressions**:
  in addition to == and != we have
  → =∼: like == but right side may contain wildcards
  → !∼: like != but right side may contain wildcards
- **Arithmetic expressions**: Similar to the C arithmetic operators. However, only integers are supported.
  Example:

```
#!/bin/bash
if [[ $1 > 0 && $(($2 % 10)) != 0 ]]; then
  echo Operands are valid
  let a = "$2 % 10"
  let r = "$(($1 * $2)) / $a"
  echo "expression value is $r"
else
  echo "Operand problem"
fi
```

- **File expressions**: **-option filename**
  The value is 1 if the selcted option is true and 0 otherwise.

  The available options are:

r	Shell has read permission
w	Shell has write permission
x	Shell has execute permission
e	file exists
O	file is owned by shell's uID (capital)
z	file exists but is of size 0
f	file is a regular file not a directory
d	file is a directory

Example:

```
#!/bin/bash
echo -n "Enter file name> "
read file
# Use elif in bash for the else if.
# >> in the example is output redirection(append).
# The ls output will be appended to the file.
if [ -w "$file" ]; then
 ls >> $file
 echo "More input has been appended"
elif [ -e "$file" ]; then
 echo "You have no write permission on $file"
else
 echo "$file does not exist"
fi
```

→ Control structures: If statement

```
if [<exp>]                                  # option 1
then
    <commands>
fi

if [<exp>]                                   # option 2
then
    <commands>
else
   <commands>
fi

if [<exp1>]                                   # option 2
then
   <commands 1>
elif [<exp2>]
   <commands 2>
else
   <commands 3>
fi
```

Example

```
#!/bin/bash
echo -n "Enter file name> "
read file
if [ ! -e $file ]; then
 echo "Sorry, $file does not exist."
elif [ ! -w $file ]; then
 echo "You have no write permission on $file''
 if [ -O $file ]; then
   chmod u+w $file #(grant write permission)
   echo "Write permission granted"
 else
   echo "Write permission cannot be granted"
   echo "because you don't own this file"
 fi
else
 ls >> $file
 echo "More input has been appended"
fi
```

- **while statement:**
  while(expr)
  do
    commandList
  done

```
#!/bin/bash                          Example
secretCode=zoom99
echo -n "Guess the code> "
read yourGuess
while [ $secretCode != $yourGuess ]; do
 echo "Good guess but wrong, try again"
 echo -n "Enter your guess> "
 read yourGuess
done
echo "BINGO!"
exit 0

while : ; do  # Syntax of an infinite loop
{code}
done
```

- **case (switch) statement: example**

```
case $choice in
 [cC])
 exec /bin/csh
 ;;
 [bB])
 exec /bin/bash
 ;;
 [kK])
 exec /bin/ksh
 ;;
 *)
 echo "Wrong choice, try again''
 read choice
 esac
```

for statements:

```
for VAR in {VAR value list}
do
  { code }
done

for (( i=0; i<5; i++ ))
do
  { code }
done
          # using command line arguments
for people in $1 $2 $3 $4
do
  echo $people
done
           # using all command line arguments
for people in $*
do
  echo $people
done
```