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

→ 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.

Selecting a Shell

Unix comes with a variety of shells. The most common are the Bourne shell (Bash), the C shell and the korn shell.
The Unix system administrator chooses a shell for each user when the account is created.
The name of your shell program can be found in corresponding line in /etc/passwd.
All shells are nomally located in /usr/bin/, they differ in their level of functionalities.

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

→ Changing your shell

You can change your shell by changing your entry in the **passwd** file.
Problem : you need to be a **super user**.
You can also change your shell by running the corresponding shell programs.
For example :
You login shell is **sh**, to run **csh**, simply type csh.

→ Determining your default shell

The full pathname of the login shell is stored in the shell variable **SHELL**.
You get your running shell by typing

```
echo $SHELL
/usr/bin/csh
```

→ 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.

Note that a shell is invoked either automatically, at the login, or manually.

→ 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 if the previous one fails.
E.g., cc prog1.c || CC prog1.c || gcc prog1.c

&& : Executes a command if the previous one succeeds
E.g., CC prog1.c -o prog1 && ./prog1

&: 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

Shells are more than command interpreters
They have their own programming languages.
A **shell script**, like a shell program, is a file that contains 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/sh:
  sh *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.

Example:

```
#!/bin/bash
#This is a sample bash script
echo -n the date of today is’ ’
# -n omits new line in bash
date
```

```
#!/bin/ksh
#This is a sample K-shell script
echo ‘‘the date of today is \c’’
# \c omits new line in K-shell
date
```

Shell variables

Two kinds of variables are supported by a shell, user-defined and shell environment variables.
Both variables store data as strings.
When a subshell is created, it gets of copy of its parent shell environment variables.

→ **Shell environment variables**
Used to customize the environment in which your shell runs.
Most of these variables are initialized by the start-up file /etc/profile.
You can customize your environment by assigning different values to these variales in your ~/.profile file.
→ **User-defined variables**
Used within shell scripts as temporary storage.

You can use the **set** command without any parameters to display all the shell variables and their values.
example, on my server account.
set

```
D        uwindsor.ca
argv     ()
cwd      /home/cs/faculty/boufama
history 40
home     /home/cs/faculty/boufama
mail     /usr/mail/boufama
path     (/usr/sbin /usr/bsd /usr/bin /bin /etc
         /usr/etc /usr/bin/X11 /usr/local/ucc/bin
         /usr/java2/bin .)
prompt   (!)%
shell    /bin/csh
status   0
term     dtterm
uid      1001
user     boufama
```

→ Some important read/write shell environment variables

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

Note: In order to access a shell variable, you must precede its name by the $ sign.
E.g., echo $MAIL

→ Some important 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

Defining an environment(global) variable

A user can also define a local shell variable that can be made a shell environment variable.
Example using bash (/bin/bash):

```
INCLUDES=$HOME/includes      #local variable
echo $INCLUDES → /users/admin256/includes
```

Because INCLUDES is local variable, it will not be accessible by subshells.
INCLUDES can be made an environment variable:
export INCLUDES

→ How to unset a variable

syntax:
unset [var-name-list]

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

```
#!/bin/bash                         Example 1
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
  → $?name: is 1 if variable is set and 0 otherwise
  Example: dir=/export/home/
  echo ”my home directory is ${dir}boufama/”
  → /export/home/boufama/

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

repeat statement: until [exp] do

```
Example:
until [ i -eq 10 ]
do
 {code}
 let i++
done
           #declare an array without initializatio
declare -a array
for name in $*; do
 array=("${array[@]}" $name)
done
echo ${array[@]} #print all array elements
i=0
until [ $i -eq $# ]; do
 echo -n ${array[$i]} #print one array element
 echo
 let i++
done
```

trap command: to handles signals in bash
You can trap any signal, except SIGKILL

```
#!/bin/bash
trap '
{
 echo "CTRL-C by user, bye bye :)"
 exit 1
}' INT
while : ;do
 echo "Infinite loop!!!!"
 sleep 5
done
```

To ignore interrupt, use ' ' INT

```
#!/bin/bash
trap '' INT
while : ;do
 echo "Infinite loop!!!!"
 sleep 5
done
```

```
trap ’’ INT
clear ; stty echo #turn off the echo mechanism
echo -n "Enter your passwd here> "
read secretPass
echo " "
echo -n "Confirm your passwd here> "
read confirmPass
echo " "
if [ "$secretPass" != "$confirmPass" ]; then
 echo "Work on your short-term memory first"
 exit 1
fi
yourGuess=""
while [ "$yourGuess" != "$secretPass" ]; do
 clear
 echo -n "Enter password to unlock screen> "
 read yourGuess
 echo " "
done
clear ; echo "You’re back in the system!"
stty echo #turn on the echo mechanism
```

Summary

- When you log onto a Unix machine, the operating system runs a program, called shell.
- A shell provides a prompt and waits for commands.
- A shell command can be internal or external.
- To execute an external command, the shell searches for its binary file in several directories, **PATH**.
- There are several shells, the most common ones are the Bourne shell, bash, Korn shell and C shell.
- bash has the built-in capability for numeric, string and file expressions.
- bash has most control structures found in a programming language
- bash has means to define and use simple variables and list variables(arrays).
- You can debug bash: *bash -x ScriptFile*