Writing Shell Scripts – part 1

CSE 2031

Fall 2010

What Is a Shell?

- A program that interprets your request to run other programs
- Most common Unix shells:
  - Bourne shell (sh)
  - C shell (csh)
  - Korn shell (ksh)
  - Bourne-again shell (bash)
- In this course we focus on Bourne shell (sh).

The Bourne Shell

- A high level programming language
- Processes groups of commands stored in files called *scripts*
- Includes
  - variables
  - control structures
  - processes
  - signals

3

Executable Files

- Contain one or more shell commands.
- These files can be made *executable.*
- # indicates a comment
  - Except on line 1 when followed by an "!"

```
% cat welcome
#!/bin/sh
echo 'Hello World!'
```

4

Executable Files: Example

```
% cat welcome
#!/bin/sh
echo 'Hello World!'
% welcome
welcome: execute permission denied
% chmod u+x welcome
% ls -l welcome
-rwxr--r-- 1 lan grad 20 Aug 29 2010 welcome
% welcome
Hello World!
% welcome > greet_them
% cat greet_them
Hello World!
```

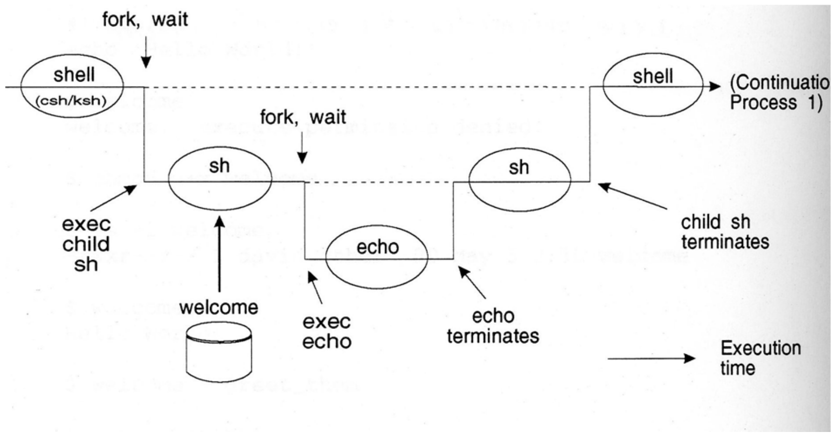
Executable Files (cont.)

- If the file is not executable, use “sh” followed by the file name to run the script.
- Example:

```
% chmod u-x welcome
% ls -l welcome
rw-r--r-- 1 lan grad 20 Aug 29 2010 welcome
% sh welcome
Hello World!
```

Processes

Consider the welcome program.



7

Processes: Explanation

- Every program is a “child” of some other program.
- Shell fires up a child shell to execute script.
- Child shell fires up a new (grand)child process for each command.
- Shell (parent) sleeps while child executes.
- Every process (executing a program) has a unique PID.
- Parent does not sleep while running background processes.

8

Variables: Three Types

- Standard UNIX variables
  - Consist of shell variables and environment variables.
  - Used to tailor the operating environment to suit your needs.
  - Examples: TERM, HOME, PATH
  - To display your environment variables, type “set”.
- User variables: variables you create yourself.
- Positional parameters
  - Also called read-only variables, automatic variables.
  - Store the values of command-line arguments.

9

User Variables

- Each variable has two parts:
  - a name
  - a value
- Syntax:

```
name=value
```

- No space around the equal sign!
- All shell variables store strings (no numeric values).
- Variable name: combinations of letters, numbers, and underscore character (_) that do not start with a number.
- Avoid existing commands and environment variables.
- Shell stores and remembers these variables and supplies value on demand.

10

User Variables (2)

- These are variables you, the user, create, read and change.
- To use a variable:

  `$varname`

- Variable substitution operator **$** tells the shell to substitute the value of the variable name.

```
#!/bin/sh
dir=/usr/include/
echo $dir
echo dir
ls $dir | grep 'ma'
```

Output:

```
/usr/include/
dir
malloc.h  math.h
numa.h  semaphore.h
```

11

echo and Variables

- What if I'd want to display the following?

```
$dir
```

- Two ways to prevent variable substitution:

```
echo '$dir'
echo \$dir
```

- Note:

`echo "$dir"` does the same as

```
echo $dir
```

12

Command Line Arguments

- Command line arguments stored in variables called positional parameters.
- These parameters are named **$1** through **$9**.
- Command itself is in parameter **$0**.
- In diagram format:

```
command arg1  arg2  arg3  arg4  arg5  arg6  arg7  arg8  arg9
   $0    $1    $2    $3    $4    $5    $6    $7    $8    $9
```

- Arguments not present get null (absence of) value

Example 1

```
% cat display_args
#!/bin/sh
echo First four arguments from the
echo command line are: $1 $2 $3 $4

% display_args William Mary Richard James
First four arguments from the
command line are: William Mary Richard James
```

Example 2

```
% cat chex
#!/bin/sh
# Make a file executable
chmod u+x $1
echo $1 is now executable:
ls -l $1

% sh chex chex
chex is now executable:
-rwx------  1 utn faculty 86 Nov 12 11:34 chex

% chex showargs
showargs is now executable:
-rwx------  1 utn faculty 106 Nov  2 14:26 showargs
```

15

Command Line Arguments (2)

- A macro is a stand-in for one or more variables
  - `$#` represents the number of command line arguments
  - `$*` represents all the command line arguments
  - `$@` represents all the command line arguments

```
% cat check_args
#!/bin/sh
echo "There are $# arguments."
echo "All the arguments are: $*"
# or echo "All the arguments are: $@"

% check_args Mary Tom Amy Tony
There are 4 arguments.
All the arguments are: Mary Tom Amy Tony
```

16

Command Line Arguments (3)

- Note: $# does NOT include the program name (unlike argc in C programs)

- What if the number of arguments is more than 9? How to access the 10th, 11th, etc.?
- Use **`shift`** operator.

17

shift Operator

- **shift** promotes each argument one position to the left.
- Operates as a conveyor belt.
- Allows access to arguments beyond $9.
  shifts contents of $2 into $1
  shifts contents of $3 into $2
  shifts contents of $4 into $3
  etc.
- Eliminates argument(s) positioned immediately after the command.
- Syntax:

```
shift        # shifting arguments one position to the left
```

- After a shift, the argument count stored in **`$#`** is automatically decremented by one.

18

Example 1

```
% cat args
#!/bin/sh
echo "arg1 = $1, arg8 = $8, arg9 = $9, ARGC = $#"
myvar=$1    # save the first argument
shift
echo "arg1 = $1, arg8 = $8, arg9 = $9, ARGC = $#"
echo "myvar = $myvar"

% args 1 2 3 4 5 6 7 8 9 10 11 12
arg1 = 1, arg8 = 8, arg9 = 9, ARGC = 11
arg1 = 2, arg8 = 9, arg9 = 10, ARGC = 10
myvar = 1
```

19

Example 2

```
% cat show_shift
#!/bin/sh
echo "arg1=$1, arg2=$2, arg3=$3"
shift
echo "arg1=$1, arg2=$2, arg3=$3"
shift
echo "arg1=$1, arg2=$2, arg3=$3"

% show_shift William Richard Elizabeth
arg1=William, arg2=Richard, arg3=Elizabeth
arg1=Richard, arg2=Elizabeth, arg3=
arg1=Elizabeth, arg2= , arg3=
```

20

Example 3

```
% my_copy dir_name filename1 filename2 filename3 …

# This shell script copies all the files to
  directory "dir_name"

% cat my_copy
#!/bin/sh
# Script allows user to specify, as the 1st argument,
# the directory where the files are to be copied.
location=$1
shift
files=$*
cp $files $location
```

21

Shifting Multiple Times

Shifting arguments three positions: 3 ways to write it

```
shift
shift
shift

shift; shift; shift

shift 3
```

22

User Variables and Quotes

`name=value`

- If `value` contains no space ⇒ no need to use quotes …

```
#!/bin/sh
dir=/usr/include/
echo $dir
```

- … unless you want to protect the literal, in which case use single quotes.

```
% cat quotes
#!/bin/sh
# Test values with quotes
myvar1=$100
myvar2='$100'
echo The price is $myvar1
echo The price is $myvar2

% quotes 5000
The price is 500000
The price is $100
```

23

User Variables and Quotes (2)

- If `value` contains one or more spaces:
- use single quotes for NO interpretation of metacharacters (protect the literal)
- use double quotes for interpretation of metacharacters

```
% cat quotes
#!/bin/sh
myvar=`whoami`
squotes='Today is `date`, $myvar.'
dquotes="Today is `date`, $myvar."
echo $squotes
echo $dquotes
% quotes
Today is `date`, $myvar.
Today is Fri Nov 12 12:07:38 EST 2010, cse12345.
```

24

Example

```
% cat my_script
#!/bin/sh
dirs='/usr/include/ /usr/local/'    # need single quotes
echo $dirs
ls -l $dirs

% my_script
/usr/include/ /usr/local/
/usr/include/:
total 2064
-rw-r--r--   1 root root   5826 Feb 21  2005 FlexLexer.h
drwxr-xr-x   2 root root   4096 May 19 05:39 GL
...
/usr/local/:
total 72
drwxr-xr-x  2 root root 4096 Feb 21  2005 bin
drwxr-xr-x  2 root root 4096 Feb 21  2005 etc
...
```

Reading User Input

- Reads from standard input.
- Stores what is read in user variable.
- Waits for the user to enter something followed by <RETURN>.
- Syntax:
  ```
  read varname        # no dollar sign $
  ```
- To use the input:
  ```
  echo $varname
  ```

Example 1

```
% cat greeting
#!/bin/sh
echo –n "Enter your name: "
read name
echo "Hello, $name. How are you today?"

% readit
Enter your name: Jane
Hello, Jane.  How are you today?
```

27

Example 2

```
% cat doit
#!/bin/sh
echo –n 'Enter a command: '
read command
$command
echo "I'm done. Thanks"

% doit
Enter a command: ls lab*
lab1.c lab2.c lab3.c lab4.c  lab5.c  lab6.c
I'm done. Thanks

% doit
Enter a command: who
lan     pts/200 Sep 1   16:23   (indigo.cs.yorku.ca)
jeff    pts/201 Sep 1   09:31   (navy.cs.yorku.ca)
anton   pts/202 Sep 1   10:01   (red.cs.yorku.ca)
I'm done. Thanks
```

28

Reading User Input (2)

- More than one variable may be specified.
- Each word will be stored in separate variable.
- If not enough variables for words, the last variable stores the rest of the line.

29

Example 3

```
% cat read3
#!/bin/sh
echo "Enter some strings: "
read string1 string2 string3
echo "string1 is: $string1"
echo "string2 is: $string2"
echo "string3 is: $string3"

% read3
Enter some strings:
This is a line of words
string1 is: This
string2 is: is
string3 is: a line of words
```

30

Next time …

- Control structures (if, for, while, …)
- Difference between `$*` and `$@`
- Shell variables

- Reading for this lecture: tutorial from "Just Enough UNIX" 5th edition by Paul K. Andersen

31