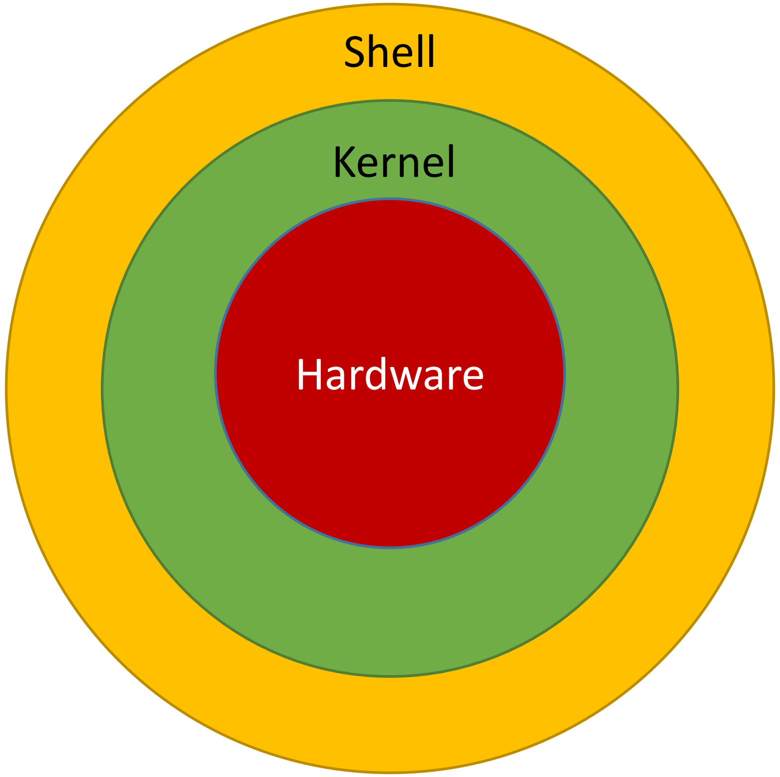
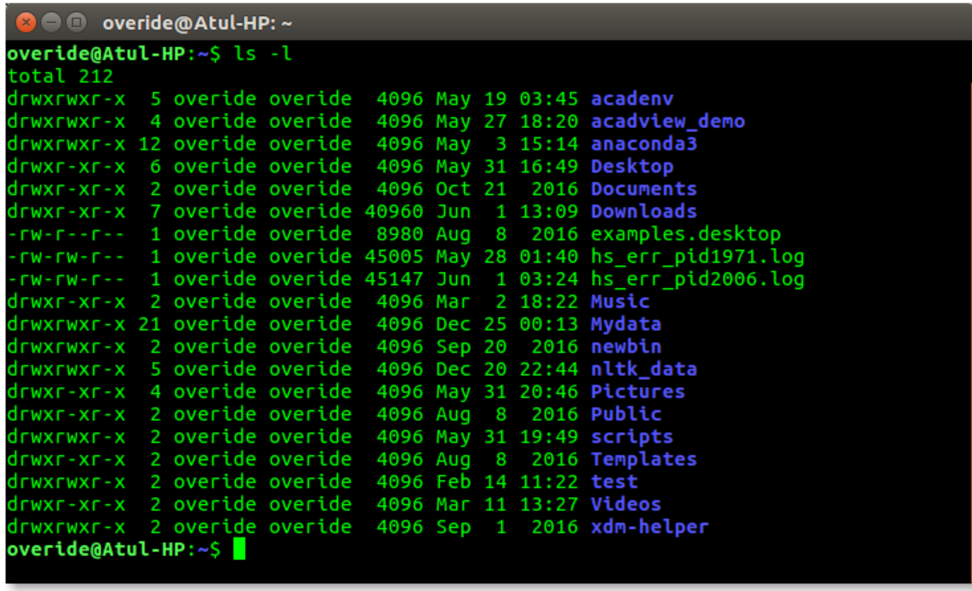
Systems programming

Basics of shell scripting

Most of the slides in this lecture are either from or adapted from the slides provided by Dr. Ahmad Barghash





What is Shell?

- Shell is
  - Command Interpreter that turns text that you type (at the command line) into actions:
  - Simply: a user Interface that takes the commands from user
- Shell scripting can do
  - Customization of a Unix session, features, or processes of OS
  - Scripting

There are many reasons to write shell scripts:

- To avoid repetitive work and automation
- System admins use shell scripting for routine backups
- System monitoring
- Adding new functionality to the shell etc.

Programming or Scripting Language?

- Shell scripting allows us to use the shell's abilities and to automate a lot of tasks that would otherwise require a lot of commands.

- **What** is the difference between programming and scripting languages:

  - Programming languages are generally a lot more powerful and a lot faster than scripting languages. Programming languages generally start from source code and are compiled into an executable. This executable is not easily ported into different operating systems.
  - A scripting language also starts from source code but is not compiled into an executable. Rather, an interpreter reads the instructions in the source file and executes each instruction.

Popular Shells

- **sh** Bourne Shell
- **ksh** Korn Shell
- **csh,tcsh** C Shell
- **bash** Bourne-Again Shell

Certainly, the most popular shell is “bash”. Bash is a compatible shell that incorporates useful features from the Korn shell (ksh) and C shell (csh)

In order to search which shell type your operating system support, you type the command:
$ cat /etc/shells

```
# /etc/shells: valid login shells
/bin/sh
/bin/bash
/bin/rbash
/bin/dash
```

The first bash program

- There are two major text editors in Linux:
  - vi/vim, emacs (or xemacs).

- So fire up a text editor; for example:

```
$ vi &      # vim TextFileName
```

and type the following inside it:

```
#!/bin/bash
echo “Hello World”
```

- The first line tells Linux to use the bash interpreter to run this script. We call it hello.sh.
- *echo* is a built-in *command* in the *bash* and *C shells* that writes its *arguments* to *standard output*.
  - Echo arguments: n (newline), t (Tab), b (Backspace), and many others

```
$ chmod 700 hello.sh  #chmod to change access permissions
$ ./hello.sh
Hello World
```

The second bash program

- We write a program that copies all files into a directory, and then deletes the directory along with its contents. This can be done with the following commands:

```
$ mkdir trash          #mkdir creates a directory in specific name
$ cp * trash           #cp copy
$ rm -rf trash         #rm remove
```

- Instead of having to type all that interactively on the shell, write a shell program instead:

```
$ cat trash.sh     # cat reads files sequentially, writing them to standard output (monitor)
#!/bin/bash
# this script deletes some files
mkdir trash
cp * trash
rm -rf trash
echo "Deleted all files!"
```

Variables

- We can use variables as in any programming languages. Their values are always stored as strings, but there are mathematical operators in the shell scripting language that will convert variables to numbers for calculations.

- We have no need to declare a variable, just assigning a value to its reference will create it.

- Example

No **whitespaces** before and after the assignment operand =.

```
#!/bin/bash
STR="Hello CS415 Students!"
echo $STR
```

- Line 2 creates a variable called STR and assigns the string "Hello CS415 Students!" to it. Then the value of this variable is retrieved by putting the '$' in at the beginning.

Important note about the assignment statement

- Unlike most modern languages, Bash is pretty picky about the syntax for setting variables. In particular, no whitespace is allowed between the variable name, the equals sign, and the value.

- All of these examples would cause Bash to throw an error:

```
var_a= "Hello World"
var_a = "Hello World"
var_a ="Hello World"
```

Valid Variable Names

- Variable names can contain a sequence of alphanumeric characters and underscores. For variables created by you, the user, they should start with either an alphabetical letter or an underscore (i.e., not a number).

- Examples of valid variable names:
  - hey
  - x9
  - GRATUITOUSLY_LONG_NAME
  - _secret

Single and Double Quote

- When assigning character data containing spaces or special characters, the data must be enclosed in either single or double quotes.

- Using double quotes to show a string of characters will allow any variables in the quotes to be resolved

$ var="test string"

$ newvar="Value of var is $var"

$ echo $newvar

Value of var is test string

- Using single quotes to show a string of characters will not allow variable resolution

$ var='test string'

$ newvar='Value of var is $var'

$ echo $newvar

Value of var is $var

The export command

- The export command put a variable into the environment so it will be accessible to child processes. For instance:

```
$ x=hello
$ bash          # Run a child shell.
$ echo $x       # Nothing in x.
$ exit          # Return to parent.
$ export x
$ bash
$ echo $x
hello           # It's there.
```

- If the child modifies x, it will not modify the parent’s original value. Verify this by changing x in the following way:

```
$ x=ciao
$ exit
$ echo $x
hello
```

Environmental Variables

- There are two types of variables:
  - Local variables
  - Environmental variables
- Environmental variables are set by the system and can usually be found by using the env command. Environmental variables hold special values. For instance:

$ echo $SHELL

/bin/bash

$ echo $PATH

/usr/X11R6/bin:/usr/local/bin:/bin:/usr/bin

- Environmental variables are defined in /etc/profile, /etc/profile.d/ and ~/.bash_profile. These files are the initialization files and they are read when bash shell is invoked.
- When a login shell exits, bash reads ~/.bash_logout

Environmental Variables

- PATH: The search path for commands. It is a colon-separated list of directories that are searched when you type a command.

- Usually, we type in the commands in the following way:

$ ./command

- By setting PATH=$PATH:. our working directory is included in the search path for commands, and we simply type:

$ command

Environment Variables

- LOGNAME: contains the user name
- HOSTNAME: contains the computer name.

- RANDOM: random number generator
- SECONDS: seconds from the beginning of the execution

Read command

- The read command allows you to prompt for input and store it in a variable.

- Example:

```
#!/bin/bash
echo -n “Enter name of file to delete: ”
read file
echo “Type 'y' to remove it, 'n' to change your mind ... ”
rm -i $file
echo "That was YOUR decision!”
```

- Line 2 prompts for a string that is read in line 3. Line 4 uses the interactive remove (rm -i) to ask the user for confirmation.

Arithmetic Evaluation

- The let statement can be used to do mathematical functions:

```
$ let X=10+2*7
$ echo $X
24
$ let Y=X+2*4
$ echo $Y
32
```

- An arithmetic expression can be evaluated by $[expression] or $((expression))

```
$ echo “$((123+20))”
143
$ VALORE=$[123+20]
$ echo “$[123*$VALORE]”
17589
```

Arithmetic Evaluation

- Available operators: +, -, /, *, %

- Example

```
$ cat arithmetic.sh
#!/bin/bash
echo -n “Enter the first number: ”; read x
echo -n “Enter the second number: ”; read y
add=$(($x + $y))
sub=$(($x - $y))
mul=$(($x * $y))
div=$(($x / $y))
mod=$(($x % $y))
# How can we print out the answers:
echo “Sum: $add”
echo “Difference: $sub”
echo “Product: $mul”
echo “Quotient: $div”
echo “Remainder: $mod”
```

Shell Basic Operators – Arithmetic Operators

Assume variable **a** holds 10 and variable **b** holds 20 then −

Operator	Description	Example
+ (Addition)	Adds values on either side of the operator	`expr $a + $b` will give 30
- (Subtraction)	Subtracts right hand operand from left hand operand	`expr $a - $b` will give -10
* (Multiplication)	Multiplies values on either side of the operator	`expr $a \* $b` will give 200
/ (Division)	Divides left hand operand by right hand operand	`expr $b / $a` will give 2
% (Modulus)	Divides left hand operand by right hand operand and returns remainder	`expr $b % $a` will give 0
= (Assignment)	Assigns right operand in left operand	a = $b would assign value of b into a
== (Equality)	Compares two numbers, if both are same then returns true.	[$a == $b] would return false.
!= (Not Equality)	Compares two numbers, if both are different then returns true.	[$a != $b] would return true.

It is very important to understand that all the conditional expressions should be inside square braces with spaces around them, for example **[$a == $b]** is **correct** whereas, **[$a==$b]** is **incorrect**.

Shell Basic Operators – Relational Operators

Assume variable **a** holds 10 and variable **b** holds 20 then –

Operator	Description	Example
-eq	Checks if the value of two operands are equal or not; if yes, then the condition becomes true.	[$a -eq $b] is not true.
-ne	Checks if the value of two operands are equal or not; if values are not equal, then the condition becomes true.	[$a -ne $b] is true.
-gt	Checks if the value of left operand is greater than the value of right operand; if yes, then the condition becomes true.	[$a -gt $b] is not true.
-lt	Checks if the value of left operand is less than the value of right operand; if yes, then the condition becomes true.	[$a -lt $b] is true.
-ge	Checks if the value of left operand is greater than or equal to the value of right operand; if yes, then the condition becomes true.	[$a -ge $b] is not true.
-le	Checks if the value of left operand is less than or equal to the value of right operand; if yes, then the condition becomes true.	[$a -le $b] is true.

It is very important to understand that all the conditional expressions should be placed inside square braces with spaces around them. For example, **[$a -lt $b]** is **correct** whereas, **[$a -lt $b]** is **incorrect**.

Shell Basic Operators – Boolean Operators

Assume variable **a** holds 10 and variable **b** holds 20 then −

Operator	Description	Example
!	This is logical negation. This inverts a true condition into false and vice versa.	[! false] is true.
-o	This is logical **OR**. If one of the operands is true, then the condition becomes true.	[$a -lt 20 -o $b -gt 100] is true.
-a	This is logical **AND**. If both the operands are true, then the condition becomes true otherwise false.	[$a -lt 20 -a $b -gt 100] is false.

Shell Basic Operators – String Operators

Assume variable **a** holds "abc" and variable **b** holds "efg" then −

Operator	Description	Example
=	Checks if the value of two operands are equal or not; if yes, then the condition becomes true.	[$a = $b] is not true.
!=	Checks if the value of two operands are equal or not; if values are not equal then the condition becomes true.	[$a != $b] is true.
-z	Checks if the given string operand size is zero; if it is zero length, then it returns true.	[-z $a] is not true.
-n	Checks if the given string operand size is non-zero; if it is nonzero length, then it returns true.	[-n $a] is not false.
str	Checks if **str** is not the empty string; if it is empty, then it returns false.	[$a] is not false.

Shell Basic Operators – File Operators

Operator	Description	Example
-b file	Checks if file is a block special file; if yes, then the condition becomes true.	[-b $file] is false.
-c file	Checks if file is a character special file; if yes, then the condition becomes true.	[-c $file] is false.
-d file	Checks if file is a directory; if yes, then the condition becomes true.	[-d $file] is not true.
-f file	Checks if file is an ordinary file as opposed to a directory or special file; if yes, then the condition becomes true.	[-f $file] is true.
-g file	Checks if file has its set group ID (SGID) bit set; if yes, then the condition becomes true.	[-g $file] is false.
-k file	Checks if file has its sticky bit set; if yes, then the condition becomes true.	[-k $file] is false.
-p file	Checks if file is a named pipe; if yes, then the condition becomes true.	[-p $file] is false.
-t file	Checks if file descriptor is open and associated with a terminal; if yes, then the condition becomes true.	[-t $file] is false.
-u file	Checks if file has its Set User ID (SUID) bit set; if yes, then the condition becomes true.	[-u $file] is false.
-r file	Checks if file is readable; if yes, then the condition becomes true.	[-r $file] is true.
-w file	Checks if file is writable; if yes, then the condition becomes true.	[-w $file] is true.
-x file	Checks if file is executable; if yes, then the condition becomes true.	[-x $file] is true.
-s file	Checks if file has size greater than 0; if yes, then condition becomes true.	[-s $file] is true.
-e file	Checks if file exists; is true even if file is a directory but exists.	[-e $file] is true.

Conditional Statements

- Conditionals let us decide whether to perform an action or not, this decision is taken by evaluating an expression.
- Conditional Statements: In total, there are 5 conditional statements which can be used in bash programming
  1. if statement
  2. if-else statement
  3. if..elif..else..fi statement (Else If ladder)
  4. if..then..else..if..then..fi..fi..(Nested if)
  5. case statement

Conditional Statement – if statement

- The block will be executed if a specified condition is true.
- Syntax:

```
if [ expression ]
then
   statement
fi
```

Put **spaces** after [and before], and around the operators and operands.

Conditional Statement – if statement (Example)

```
#!/bin/bash

echo -n "Enter a number: "
read VAR

if [[ $VAR -gt 10 ]]
then
  echo "The variable is greater than 10."
fi
```

Conditional Statement – if…else statement

- If specified condition is not true in if part then else part will be executed.
- Syntax:

```
if [ expression ]
then
   statement1
else
   statement2
fi
```

Conditional Statement – if…else statement (Example)

```
#!/bin/bash

echo -n "Enter a number: "
read VAR

if [[ $VAR -gt 10 ]]
then
  echo "The variable is greater than 10."
else
  echo "The variable is equal or less than 10."
fi
```

Conditional Statement – if..elif..else..fi statement

- To use multiple conditions in one if-else block, then elif keyword is used in shell. If expression1 is true then it executes statement 1 and 2, and this process continues. If none of the condition is true then it processes else part.
- Syntax:

```
if [ expression1 ]
then
   statement1
   statement2
   .
   .
elif [ expression2 ]
then
   statement3
   statement4
   .
   .
else
   statement5
fi
```

Conditional Statement – if..elif..else..fi statement (example)

```
#!/bin/bash

echo -n "Enter a number: "
read VAR

if [[ $VAR -gt 10 ]]
then
  echo "The variable is greater than 10."
elif [[ $VAR -eq 10 ]]
then
  echo "The variable is equal to 10."
else
  echo "The variable is less than 10."
fi
```

Conditional Statement – Nested if

- Bash allows you to nest if statements within if statements. You can place multiple if statement inside another if statement.

Find the maximum value among three given values

```
#!/bin/bash

echo -n "Enter the first number: "
read VAR1
echo -n "Enter the second number: "
read VAR2
echo -n "Enter the third number: "
read VAR3
```

```
if [[ $VAR1 -ge $VAR2 ]]
then
  if [[ $VAR1 -ge $VAR3 ]]
  then
    echo "$VAR1 is the largest number."
  else
    echo "$VAR3 is the largest number."
  fi
else
  if [[ $VAR2 -ge $VAR3 ]]
  then
    echo "$VAR2 is the largest number."
  else
    echo "$VAR3 is the largest number."
  fi
fi
```

Conditional Statement – case statement

- Case statement works if specified value match with the pattern then it will execute a block of that particular pattern
- When a match is found all of the associated statements until the double semicolon (;;) is executed.
- a *) is used to accept any value not matched with list of values
- Syntax:

```
case  in
   Pattern 1) Statement 1;;
   Pattern n) Statement n;;
   *) statements;;

esac
```

Conditional Statement – case statement - example

```
CARS="bmw"

#Pass the variable in string
case "$CARS" in
  #case 1
  "mercedes") echo "Headquarters - Affalterbach, Germany" ;;

  #case 2
  "audi") echo "Headquarters - Ingolstadt, Germany" ;;

  #case 3
  "bmw") echo "Headquarters - Chennai, Tamil Nadu, India" ;;
esac
```

Example (case.sh)

```
$ cat case.sh
#!/bin/bash
  echo -n “Enter a number 1 < x < 10: ”
 read x
  case $x in
        1) echo “Value of x is 1.”;;
        2) echo “Value of x is 2.”;;
        3) echo “Value of x is 3.”;;
        4) echo “Value of x is 4.”;;
        5) echo “Value of x is 5.”;;
        6) echo “Value of x is 6.”;;
        7) echo “Value of x is 7.”;;
        8) echo “Value of x is 8.”;;
        9) echo “Value of x is 9.”;;
        0 | 10) echo “wrong number.”;;
        *) echo “Unrecognized value.”;;
  esac
```

Iteration Statements

- The for structure is used when you are looping through a range of variables.

```
for var in list
  do
    statements
  done
```

- statements are executed with var set to each value in the list.

- Example

```
#!/bin/bash
  let sum=0
  for num in 1 2 3 4 5
    do
      let “sum = $sum + $num”
    done
  echo $sum
```

15

Iteration Statements

```
#!/bin/bash
  for x in paper pencil pen
 do
   echo "The value of variable x is: $x"
   sleep 1
  done
```

"The value of variable x is: paper"
"The value of variable x is: pencil"
"The value of variable x is: pen"

- if the list part is left off, var is set to each parameter passed to the script ($1, $2, $3,…)

```
$ cat for1.sh
  #!/bin/bash
  for x
  do
   echo "The value of variable x is: $x"
   sleep 1
  done
$ ./for1.sh arg1 arg2
```

The value of variable x is: arg1
The value of variable x is: arg2

Special Shell Parameters

$# is the number of parameters passed

$0 returns the name of the shell script running as well as its location in the file system

$* gives a single word containing all the parameters passed to the script

$@ gives an array of words containing all the parameters passed to the script

```
$ cat sparameters.sh
#!/bin/bash
echo “$#; $0; $1; $2; $*; $@”
$ ./sparameters.sh arg1 arg2
2; ./sparameters.sh; arg1; arg2; arg1 arg2; arg1 arg2
```

Trash

$ cat trash.sh

```
#!/bin/bash
if [ $# -eq 1 ];
then
	if [ ! –d “$HOME/trash” ];
	then
		mkdir “$HOME/trash”
	fi
	mv $1 “$HOME/trash”
else
	echo “Use: $0 filename”
	exit 1
fi
```

```
./trash.sh
Use: ./trash.sh filename
```

Example (old.sh)

```
$ cat old.sh
#!/bin/bash
# Move the command line arg files to old directory.
if [ ! –d "$HOME/old" ]
then
  mkdir "$HOME/old"
fi
echo "The following files will be saved in the old directory:"
echo $*
for file in $* #loop through all command line arguments
do
  mv $file "$HOME/old/"
  chmod 400 "$HOME/old/$file"
done
ls -l "$HOME/old"
```

Converting the access mode to each file into "read-only".

ls -l list in long format showing permissions

Using Arrays with Loops

- In the bash shell, we may use arrays. The simplest way to create one is using one of the two subscripts:

```
pet[0]=dog
pet[1]=cat
pet[2]=fish
pet=(dog cat fish)
```

- We may have up to 1024 elements. To extract a value, type ${arrayname[i]}

```
$ echo ${pet[0]}
  dog
```

- To extract all the elements, use an asterisk as:

```
echo ${arrayname[*]}
```

- We can combine arrays with loops using a for loop:

```
for x in ${arrayname[*]}
  do
        ...
  done
```

A C-like for loop

- An alternative form of the for structure is

```
for (( EXPR1 ; EXPR2 ; EXPR3 ))
   do
         statements
   done
```

- First, the arithmetic expression EXPR1 is evaluated. EXPR2 is then evaluated repeatedly until it evaluates to 0. Each time EXPR2 evaluates to a non-zero value, statements are executed and EXPR3 is evaluated.

$ cat for2.sh

```
#!/bin/bash
echo –n “Enter a number: ”; read x
let sum=0
for (( i=1 ; $i<$x ; i=$i+1 )) ; do
  let “sum = $sum + $i”
done
echo “the sum of the first $x numbers is: $sum”
```