Overview

The shell is one of the main interfaces to UNIX that a Systems Administrator uses.

- Interpreter
  - Process the commands you enter
- Programming language
  - Process groups of commands stored in a file called shell scripts.
  - Like other languages, shells have
    - Variables
    - Control flow commands

Various Shells

- There are many different types
  - Bourne shell, sh
    The original shell from AT&T.
  - Korn Shell, ksh
    A superset of Bourne shell that lets you edit the command line.
  - C shell, csh
    Shell for BSD UNIX. Which uses C syntax and has many conveniences.
  - modern updates – bash, tcsh
    modern updates of the Bourne and C shell
    bash is the default Linux shell.
    - **Most examples used in this lecture are from bash.**

Shells are programs

- Shells are just executable programs.
  - Different shells use different syntax and provide different services.
- You can start any shell at anytime
  - Example:

    $sh

    $csh

    $exit
- You exit a shell with logout, exit or *CTRL-D*
- Each user has an entry in the /etc/passwd file which includes the name of the shell to execute.
  - You can change your login shell using the chsh command.
  - The file /etc/shells contains a list of valid shells.

3

Executing Commands

- As a command interpreter, the shell performs the following tasks.
  1. Wait for the user to enter a command
  2. Parse the command line
     - **This is the step this lecture concentrates on**
  3. Find the executable file for the command
     1. a shell function
     2. a built-in shell command
     3. an executable program.
  4. If the command can't be found generate an error message
  5. If it is found, fork off a child process to execute the command
  6. Wait until the command is finished
  7. Return to step 1

4

Parsing the command line

- When parsing the command line, bash performs three major steps
  - I/O redirection
  - Expansion
    - Tilde ~, Variable, Command, Arithmetic, Filename , Brace {} etc.
  - Remove quote characters
- Much of the above process is achieved using characters with special meanings.

Special Character

Characters	Meaning
white space	spaces, tabs are used to separate arguments on the command line
newline character	indicates the end of a command
' " \	quote characters which change the way the shell interprets special characters
& && || ;	Control the ways in which commands are executed
< > << >> |	I/O redirection characters
* ? [	filename substitution characters

Using Special Characters

- When you enter a command, the shell searches for special characters, and it then performs some special tasks to replace the special characters.
  - Example:
    - echo *
      will not display a *.
    - To actually use a special character (e.g. *) as itself you have to quote it
      - using a pair of single quotes. echo '*'
      - using a single back slash. Echo *

Using Special Characters

- Check the actual command after the replacing:
  - Turn on: set -x
  - Turn off: set +x
  - Example:

```
sh-2.05b$ set -x
sh-2.05b$ echo *
+ echo a
a
sh-2.05b$ echo '*'
+ echo '*'
*
sh-2.05b$ echo \*
+ echo '*'
*
sh-2.05b$ set +x
```

Quote Characters

Quote Character	Name	Purpose
'	single quote	used in pairs, removes the special meaning of all characters within the pair of single quotes
\	back slash escape character	removes special meaning from the next character
"	Double quote	must be used in pairs, removes special meaning from all characters in the pair except $ ' \

Quote Characters

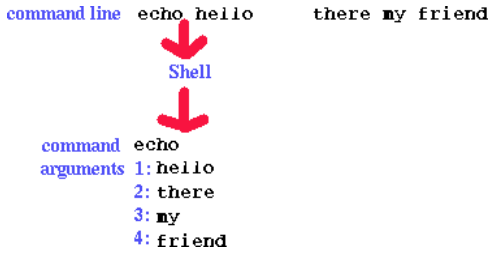
- Keeping track of special characters and their meanings can be a hassle

```
[root@faile 6]# echo a \" is a special character
a " is a special character
[root@faile 6]# echo a " is a special character
>
>here it is waiting for a closing "
a is a special character

here it is waiting for a closing
[root@faile 6]# echo a "" is a special character
a " is a special character
[root@faile 6]# echo showing multiple \\\\\\\\ is tricky
showing multiple \\\\ is tricky
 [root@faile 6]# echo then again maybe not '\\\\'
then again maybe not \\\\
[root@faile 6]# echo then again maybe not "\\\\"
then again maybe not \\
```

Commands and arguments

- Any command line consists of
  - Command and any number of arguments
  - All separated by white space
  - When the shell parses the command line it removes the white space



Commands and Arguments

- White space is a special character
  - Quote it
  - Example:

```
bash-2.05b$ echo Hello there
+ echo Hello there
Hello there
bash-2.05b$ echo Hello     there
+ echo Hello there
Hello there
bash-2.05b$ echo "Hello     there"
+ echo 'Hello     there'
Hello     there
```

Control Operator

- Control operators are a collection of characters (; & && ||) which change the operation of the command line.
  - The ; character can be used to place more than one command on a single line.
    - Example: ls ; echo now in root directory ; cd / ; ls
  - Place the command in the background: &
    - Example: firefox &

13

Control Operator

- **&&** characters tell the shell to execute the 2nd command only if the first command has an exit status 0 (and)
  - Example:

    grep root /etc/passwd && echo it works

    grep not_there /etc/passwd && echo it works
- **||** characters tell shell to execute the 2nd command only if the first command has an exit status of not 0
  - Example:

    grep root /etc/passwd || echo it does not work

    grep not_there /etc/passwd || echo it does not work
- Example: On Linux, /etc/init.d/sshd has a line

  $SSHD $OPTIONS && success || failure

14

Command Separation and Grouping

- How to type a long command
  - Continue a Command using \ at the end of line
    - NEWLINE is the end of a command
    - \ escape the meaning of the next character
- Group Commands ()
  - User parentheses to group commands
  - The shell create a subshell for each group
    - Each subshell has its own env
    - Ex:
      - #pwd; (cd /); pwd
      - #pwd; cd /; pwd
      - (cd $1; tar –cf - .) | (cd $2 ; tar –xvf -)

15

What is I/O Redirection

- A way to send input and output (I/O) to different places.
- Used to
  - Read data from a file
    - EX: cat < /etc/passwd
  - Write data to a file
    - EX: ls –lR > all.my.files
  - Join Unix commands together
    - EX: grep "/bin/sh" /etc/passwd | cut –f1 –d:

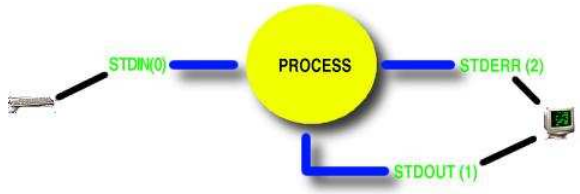
16

How does I/O redirect work

- Every process has a table of file descriptors (one for each file).
- Every process has three standard file descriptors which are used by default.

Name	File descriptor	Default destination
Standard input (stdin)	0	The Keyboard
Standard output (stdout)	1	The screen
Standard error (stderr)	2	The screen

How does I/O redirect work



- Not all commands use input and output
  - Ex: rm
- Others use all three
  - Ex: cat grep sed

I/O Special Characters

- Redirect I/O by using some special characters.

I/O Redirection	Result
Command < file	Take standard input from file
Command > file	Place output of command into file. Overwrite anything already in the file.
Command >> file	Append the output of command into file
Command << file	Take standard input for command from the following lines until a line that contains label by itself. Called a **here document**
Command1 | command2	pass the output of command1 to the input of `command2`
Command1 >& file_descriptor	redirect output of command1 to a file_descriptor (the actual number for the file descriptor)

Example using I/O redirection

- Redirecting I/O to files
  - Create a file called all.my.files which contains the output of the ls command
  - Add the message "END OF FILE" to the end of all.my.files
- Pipelines
  - Count the number of directories in the current directory
  - Display the user account name in /etc/passwd in caps.

Example using I/O redirection

- Descriptor (this works in BASH)
  - Generate the "ls –R /" output to all.my.files and error to /tmp/errors
    - $ls –lR / > all.my.files 2> /tmp/errors
  - Generate the "ls –R / " output and error output in output.and.errors
    - $ls –R / > output.and.errors 2>& 1
- Here documents

```
$cat << the_end
>This is a test of output of ls
>`ls`
>the_end
```

- If no input file is provided, some commands will wait for the input from keyboard, until ctrl-D (EOF) is pressed.
  - cat, awk, grep, mailx, …

21

Device files and I/O redirection

- UNIX treats everything (devices, processes, the kernel) as files
- Sending information to these files sends the information to the device.
  - Example:
    - The tty command displays the device file for your current terminal.

      $tty
    - Send the contents of a file to terminal 1

      $ls > /dev/tty4
    - Send a file to the bit bucket in the sky

      $cat file > /dev/null

22

Example: root cron jobs

```
#
#15 2 * * * /usr/sbin/quot /fortran > /fortran/disk.usage 2>&1
#
# check quotas once every hour
# quotacheck is run on demand by e-mailing quotacheck@cslserver from login
# scripts now so only run quotacheck once a day
#
#30 3 * * * /usr/sbin/quotacheck /fortran > /dev/null 2>&1
#
# delete old apache logs and compress new ones
#
0 1 * * * /usr/bin/find /var/adm-httpd -type f -mtime +14 -name "*log*gz" -exec
    /usr/bin/rm {} \; > /dev/null 2>&1
5 1 * * * /usr/bin/find /var/adm-httpd -type f -mtime +1 -name "*log*" \! -name "*gz"
    -exec /usr/bin/gzip {} \; > /dev/null 2>&1
#
# check number of files in /var/spool/mqueue for CEC nagios monitor
#
0,2,4,6,8,10,12,14,16,18,20,22,24,26,28,30,32,34,36,38,40,42,44,46,48,50,52,54,56,5
    8 * * * * /usr/bin/ls /var/spool/mqueue | /usr/bin/wc -l | /usr/bin/awk '{print $1}' -
    >/usr/host/nagios/mqf 2>/dev/null
```

23

Exercise

- Log in to the general lab machine
  - Redirect the standard output
    - Find all the student's home writable by the world
    - Redirect the above output file to /tmp/openhomedir.lst
  - Redirect the error output
    - Run the command "ls /etc/hosts /etc/host"
      - Redirect all the output from standard output and standard error to /tmp/ls.output.

24

Shell Variables

- Shells: the customization of your UNIX session
  - A shell defines variables to control the behavior of your UNIX session
  - Ways to use shell variables include
    - assigning values (setting)
      variable_name=[*value*]
      - Ex: name="david jones"
    - Using the value
      - When the shell sees a dollar sign ($) followed by a variable name it will replace it with its value
      - Ex: echo $name
    - What variables are set
      - The set command (actually a built-in shell command) will show all the current shell variables and their values.

25

Shell variables

- Shell variable names
  - must start with a letter or underscore character
  - followed by 0 or more letters, numbers or underscores
  - The use of the braces { } to separate variable names from other letters
  - For example

```
[david@faile 5]$ name=
[david@faile 5]$ name2=' '
[david@faile 5]$ echo fred${name}fred
fredfred
[david@faile 5]$ echo fred${name2}fred
fred fred
[david@faile 5]$ echo fred${name3}fred
fredfred
```

26

Local vs. environment variables

- By default, variables are local.
- To turn a local variable to an environment variables

  $export variablename

  Ex:

  $CLASSPATH=$NETSCAPE_HOME/classes

  $export CLASSPATH
- Environment variables are passed to sub-processes. Local variables are not.
  - Check env: env
  - Check all variables: set
- To unexport a variable

  $unset

27

Environment Control

- variables are also used for environment control.
- When a shell runs it sets up an environment in which you execute programs
- The characteristics of this environment are stored in pre-defined shell variables.

Variable Name	Purpose
HOME	The user's login directory
SHELL	The current shell being used
UID	Your user id
PATH	The executable path
PS1	The value of the command prompt

28

Startup files

- Bourne shell
  - System wide /etc/profile
  - .profile in your home dir
  - Example: /etc/profile in undergrad lab.
- Make the change take effect
  - Log out and log back in
  - Running .profile with . (DOT) built-in

    Ex: $. .profile
    - Command runs the script as the part of the current process
    - Changes will affect the login shell.
    - If without the first ., the new variable would be in effect only in the subshell running the script.

29

Environment Control

- You can change environment variables just like you would any other.

```
[root@faile 5]# PS1='fred >'
fred >echo hello
hello
fred > PATH=
fred > ls
bash: ls: No such file or directory
fred > PATH='… …'
fred > ls
Image5.gif                io3.xcf
```

  - Exercise: Change your prompt for the bash
    - Check man page, looking for PS1
    - Set prompt to be like [username@hostname:pwd%]
  - Question:
    - What is the value of PATH for your lab account? Where did it get its value?

30

Advanced variable substitution

Construct	Purpose
${variable:-value}	replace this construct with the variable's value if it has one, if it doesn't, use value but don't make variable equal to value
${variable:=value}	same as the above but if variable has no value assign it value
${variable:?message}	replace the construct with the value of the variable if it has one, if it doesn't then display message onto stderr if message is null then display prog: variable: parameter null or not set on stderr
${variable:+value}	if variable has a value replace it with value otherwise do nothing

31

Variable substitution example

- Example

```
dinbig:~$ myName=

dinbig:~$ echo my name is ${myName:-"NO NAME"}
my name is NO NAME

dinbig:~$ echo my name is $myName
my name is

dinbig:~$ echo my name is ${myName:="NO NAME"}
my name is NO NAME

dinbig:~$ echo my name is $myName
my name is NO NAME
```

32

Variable substitution example

```
dinbig:~$ herName=

dinbig:~$ echo her name is ${herName:?"she hasn't got a name"}
bash: herName: she hasn't got a name

dinbig:~$ echo her name is ${herName:?}
bash: herName: parameter null or not set
```

33

Command substitution

- Command substitution allows you to insert the output of one command into the command line of another.
  - command substitution uses back quote character `.
    - **BE CAREFUL:** Many people confuse the back quote with the single quote character. They are different.
- Exercise:
  - Create a file called hostname.log
  - Create a file named yyyy_mm_dd.log

34

Pathname expansion

- A way of specifying a number of filenames with a small number of characters.
  - Also known as filename substitution or globbing
  - Example
    - rather than
      ls -l word.doc fred.doc chap1.doc chap2.doc
    - easier to use
      ls -l *.doc
  - Some special characters are used to specify which filenames to match.
    - *
      - Any string including the null string
    - ?
      - any single character
    - [ ]
      - Matches any one character within the [ ]
        Two characters separated by a - indicates a range
        If ! or ^ are the first character then any character NOT between [ ] are matched.

35

Other expansion

- bash supports a number of other expansions which can be useful
  - brace expansion
    - Similar to pathname expansion but doesn't match real filenames.
    - Example
      ls -ld /etc/rc.d/{init,rc1,rc2}.d
  - tilde expansion
    - The ~ (tilde) character expands to either user's home directory, current working directory or previous working directory depending on following character.

36

Other expansion

- tilde expansion
  - Examples

    echo my home directory is ~

    echo working directory is ~+

    echo previous working directory is ~-
- arithmetic expansion
  - Evaluation of arithmetic expressions.
  - Examples

    echo $[(5+4)-3*2]

37

Create a shell script

- Why shell script?
  - Simply and quickly initiate a complex series of tasks or a repetitive procedure.
- Creating a simple shell script
  - A shell script is a file that contains commands that the shell can execute.
    - Any commands you enter in response to a shell prompt.
      - A utility
      - A compiled program
      - Another shell script
    - Control flow commands

38

How to Run a shell script – Method 1

- Make the file executable
  - When you create a shell script using a editor, does it have execute permission typically?
    - Example

```
[ruihong@dafinn ~/cs3451]$ ./test
./test: Permission denied.
[ruihong@dafinn ~/cs3451]$ ls -l test
-rw-------    1 ruihong  csdept        22 Jan 28 09:33 test
[ruihong@dafinn ~/cs3451]$ chmod +x test
[ruihong@dafinn ~/cs3451]$ ./test
this is a test
```

39

How to Run a shell script – Method 1

- Enter the script filename as the command
  - The shell forks a process
    - Which creates a duplicate of the shell process (subshell)
  - The new process attempt to exec the command
    - If the command is a executable program
      - Exec succeeds
      - System overlays the newly created subshell with the executables programs
    - If the the command is a shell script
      - Exec failed
      - The command is assumed to be a shell script
      - The subshell runs the commands in the shell one after another

40

How to Run a shell script – Method 2

- Pass the shell script as a parameter to a shell program
  - Run a shell script which does not have execution permission
    Ex: $sh filename
  - Run the script with different shell other than your interactive shell
    Ex: $ksh filename

41

Define the shell type in the script

- Put special characters on the first line of a shell script
  - To tell OS checks what kind of file it is before attempting to exec it
  - To tell which utility to use (sh, csh, tcsh, …)
    - The firsts two character of a script are #!
    - Then followed by the absolute pathname of the program that should execute the script

```
sh-2.05b$ more /etc/init.d/sshd
#!/bin/bash
#
# Init file for OpenSSH server daemon
#
```

42

Make a comment

- Comments make shell scripts easier to read and maintain
- Pound sign (#) start a comment line until the end of that line, except
  - #! In the first line.
  - Or inside quotes

43

Parameters and Variables

- A shell parameter is associated with a value that is accessible to the user.
  - Shell variables
    - Keyword shell variables
      - Has special meaning to the shell
      - Being created and initialized by the startup file
    - User created variables (create and assign value)
      - Local variables
      - Environment variables
  - Positional parameters
    - Allow you to access command line arguments
  - Special parameters
    - Useful others

44

Positional Parameters

- The command name and arguments
  - Why are they called positional parameters?
    - Because they can by referenced by their position on the command line
      - $0 : Name of the calling program
      - $1 - $9 : Command-line Arguments
        - The first argument is represented by $1
        - The second argument is represented by $2

45

Positional Parameters

- Example:

```
[ruihong@dafinn ~/cs3451]$ more display_5args
echo you are running script $0 with parameter $1 $2 $3 $4 $5

[ruihong@dafinn ~/cs3451]$ ./display_5args 1 2 3 4 5
you are running script ./display_5args with parameter 1 2 3 4 5
```

46

Positional Parameters

- How to access the parameter after $9?
  - shift
    - Built-in command shift promotes each of the command-line arguments.
      - The first argument (which was $1) is discarded
      - The second argument (which was $2) becomes $1
      - The third becomes the second
      - And so on
- Repeatedly using shift is a convenient way to loop over all the command-line arguments

47

Positional Parameters

- Example:

[ruihong@dafinn ~/cs3451]$ more ./demo_shift

```
echo $1 $2 $3
shift
echo $1 $2
shift
echo $1
```

[ruihong@dafinn ~/cs3451]$ ./demo_shift 1 2 3

1 2 3

2 3

3

48

```
[ruihong@dafinn ~/cs3451]$ more demo_shift
echo $1 $2 $3
shift
echo $1 $2
shift
echo $1
shift
echo $?
shift
echo $?
shift
echo $?

[ruihong@dafinn ~/cs3451]$ ./demo_shift 1 2 3
1 2 3
2 3
3
0
1
1
```


Positional Parameters

- Initialize arguments outside of the command line
  - set (sh/ksh only)
    - Set the positional parameters starting from $1, …
- Use quote for variable reference
  - Example:
    - what's the difference if $1 is null

      $ display_4args $1 a b c d

      $ display_4args "$1" a b c d

    - What will happen if a is null

```
if [ $a = 3 ]; then
  echo a is 3
fi
```

Special Parameters

- Special parameters give you some useful values
  - Number of the Command-line arguments
  - Return status of the execution of shell commands
  - Etc..
- Special parameters' value can not be changed directly, like positional parameters

51

Special Parameters $* $@

- Value of Command-line arguments: $* and $@
  - $* and $@ represent all the command_line arguments (not just the first nine)
  - "$*" : treat the entire list of arguments as a single argument
  - "$@" : produce a list of separate arguments.

52

```
sh-2.05b$ more for_test
echo "using \$@ "
for arg in "$@"
do
        echo "$arg"
done

echo "using \$* "
for arg in "$*"
do
        echo "$arg"
Done

sh-2.05b$ ./for_test 1 2 3
using $@
1
2
3
using $*
1 2 3
```

53

Special Parameters $#

- The number of arguments: $#
  - Return a decimal number
  - Use the test to perform logical test on this number

```
sh-2.05b$ more num_args
echo this script is called with $# arguments.
sh-2.05b$ ./num_args
this script is called with 0 arguments.
sh-2.05b$ ./num_args 1
this script is called with 1 arguments.
sh-2.05b$ ./num_args alice in wonder land
this script is called with 4 arguments.
```

54

Special Parameters $$ $!

- The current shell's PID number: $$
  - Ex:

```
sh-2.05b$ echo $$
11896
sh-2.05b$ echo "today is `date`" >> $$.memo
sh-2.05b$ more $$.memo
today is Sun Jan 30 00:18:09 EST 2005
```

- The PID number of last process that you ran in the background: $!
  - Ex:

```
sh-2.05b$ sleep 1000 &
[1] 11962
sh-2.05b$ echo $!
11962
```

55

Special Parameters $?

- Exit status: $?
  - When a process stops executing for any reason, it returns an exit status to its parent process.
  - By convention,
    - Nonzero represents a false value that the command failed.
    - A zero value is true and means that the command was successful
  - You can specify the exit status that a shell script returns by using the exit built-in followed by a number
    - Otherwise, the exit status of the script is the exit status of the last command the script ran.

56

```
sh-2.05b$ ls a
ls: a: No such file or directory
sh-2.05b$ echo $?
1
sh-2.05b$ echo tttt
tttt
sh-2.05b$ echo $?
0

sh-2.05b$ more exit_status
echo this program will have the exit code of 8.
exit 8
sh-2.05b$ ./exit_status
this program will have the exit code of 8.
sh-2.05b$ echo $?
8
sh-2.05b$ echo $?
0
```

57

Summary

- A shell is both a command interpreter and a programming language
- Job control
  - Control-z/fg/bg/&
- Variables
  - Local and environment variables
  - Declare and initialize a variable (no type)
  - Export unset
- Command line expansion
  - Parameter expansion/variable expansion/command/substitution/pathname expansion
  - Quote (` ` " " \ )
    - " " all but parameter, variable expansion and \
    - ` ` suppress all types of expansion
    - \ escaping the following special character

58

Summary

- Shell parameters
  - HOME
  - PATH
  - PS1
  - SHELL
  - $0
  - $n
  - $*
  - $@
  - $#
  - $$
  - $!
  - $?

59

Summary

- Special Characters
  - NEWLINE
  - ;
  - ()
  - &
  - |
  - >
  - >>
  - <
  - <<

60

Summary

- *
- ?
- \
- `
- "
- ` `
- []
- $
- .
- #
- &&
- ||
- !

61