Lecture 17

Unix Commands and Shell Scripting

Shibo Li

shiboli@cs.fsu.edu

Department of Computer Science
Florida State University

The slides are mainly from Sharanya Jayaraman

Bundling Files into an Archive

- `Tar` is a utility for creating and extracting archives. It is very useful for archiving files on disk, sending a set of files over the network, and for compactly making backups
- General Syntax: `tar options filenames`
- Commonly Used Options:
  - -c insert files into a tar file
  - -f use the name of the tar file that is specified
  - -v output the name of each file as it is inserted into orextracted from a tar file
  - -x extract the files from a tar file
  - -t list contents of an archive
  - -z gzip / gunzip if necessary

Creating an Archive with tar

- The typical command used to create an archive from a set of files is illustrated below.
- Note that each specified filename can also be a directory.
- `Tar` will insert all files in that directory and any subdirectories.The `f` flag is used to create an archive file with a specific name (usually named with an extension of .tar).

Creating an Archive with tar

- Examples
  - `tar -cvf proj.tar proj` If proj is a directory this will insert the directory and all files and subdirectories (recursively) with the name of the archive being proj.tar Note that if proj.tar already existed it will simply be overwritten; previous information will be lost.
  - `tar -cvf prog.tar *.c *.h` All files ending in .c or .h will be archived in prog.tar

Extracting files from a tar archive

- The typical tar command used to extract the files from a tar archive is illustrated below.
- The extracted files have the same name, permissions, and directory structure as the original files.
- If they are opened by another user (archive sent by email) the user id becomes that of the user opening the tar archive.
- Examples:
  - `tar -xvf proj.tar` Extract all files from proj.tar into the current directory. Note that proj.tar remains in the current directory
  - `tar -xvzf proj.tar.gz` Extract files but also unzip files in the process

The gzip/gunzip command

- this is one of the compression utilities that reduces the size of a file to take up less space on your drive. It should be used with some care.
- The compressed file has an extension of `.gz`
- Examples
  - `gzip bigfile` compresses file, flag -v can be used for verbose mode to give information. Note: original file is gone!
  - `gzip -d bigfile.gz` Restores a .gz file
  - `gunzip bigfile.gz` Same as above

The `diff` command

- `diff` compares two text files (can also be used on directories)and prints the lines for which the files differ.
- Syntax: `diff [options] file1 file2`
- Some options:
  - -b Treats groups of spaces as one
  - -i Ignores case
  - -r Includes directories in comparison
  - -w Ignores all spaces and tabs
- Example: `diff -w testprog1.c testprog2.c`

The `cmp` command

- Compares two files byte by byte and tells you where they differ.
- Generally used for binary and executable files as opposed todiff which is used for text files.
- Syntax: `cmp options file1 file2`
- Example: `cmp myfile1.o myfile2.o`

The grep command

- `grep` is a very useful utility that searches files for a particular pattern.
- The pattern can be a word, a string enclosed in single quotes,or a regular expression.
- Syntax: `grep options pattern files`
- grep has many options; a few are noted below
  - -i Ignore case
  - -n Display line numbers
  - -l Display only the names of the files and not the actual lines
  - -P pattern is a Perl regular expression, not a Unix regular expression

Some grep Examples

- `grep int *.c` find all occurrences of the pattern int in all files with a .c extension
- `grep 'main()' testprog1.c` enclosing the pattern in quotes is useful when using special characters
- `grep 'm.*n' myfile` The . matches a single character, the .* matches any number of characters; this finds anything starting with an m and ending with an n

Some grep Patterns

- Bracketed Expressions
  - `[1357]` matches 1 or 3 or 5 or 7
  - `[^1357]` matches 1 or 3 or 5 or 7
- Range Expressions: `[b-g]` matches b, c, d, e, f, g
- Named classes of expressions `[:digit:]`, `[:alnum:]`
- Special symbols
  - ? The preceding item is optional and matched at most once.
  - * The preceding item will be matched zero or more times.
  - + The preceding item will be matched one or more times.
  - . This matches any single character

More `grep` expressions

- Matching at the beginning and end
  - ^ matches the beginning of the line, thus ^#include would macth any lines with a #include at the beginning of the line.
  - $ matches the end of line
  - \< matches the beginning of a word
  - \> matches the end of a word
- | The or operator: `grep cat | dog`

Unix manpages

- The man command is used to display the manual for any ofthe Unix utilities/commands - also called a manpage
- The manual includes a description of the command, the options, the exit status(es), return value(s), errors, files,actions, examples, etc.
- Syntax: `man command`
- The man command also comes with a few options:
  - To show a certain section of the manual: `man section_number command`
  - To show only the synopsis of the command: `man -f command`
  - To show all the manuals that contain a reference to a command: `man -k command`

Shell Scripts

- A Shell Script is an executable file containing
  - Unix shell commands
  - Programming control constructs (if, then, while, until, case,for, break, continue, while, *etc.*)
  - Basic programming capabilities (assignments, variables, arguments, expressions, *etc.*)
- The file contents comprise the script

Shell Scripts

- Unlike a C++ program, that is compiled and then executed, shell scripts are **interpreted**.
- Usually, the first line of the script indicates which shell is used to interpret the script.

You First Script

```
#!/bin/sh
#this is the script in file firstscript.sh cal
date
who | grep shiboli
exit
```

- The ‘‘`#!`’’ is used to indicate that what follows is the shell used to interpret the script
- The ‘‘`exit`’’ command immediately quits the shell script (by default it will also quit at the end of the file)

Executing a shell script

- `sh myscript #uses Bourne shell`
- `tcsh myscript #uses tcshell`
- Note that the above explicitly invoke the appropriate shell with the file containing the commands as a parameter.
- You can also make the file executable and then simple run as a command

```
>chmod 755 myscript (or chmod +x myscript)
>myscript
```

Shell Scripts

- Advantages
  - Can quickly setup a sequence of commands to avoid a repetitive task
  - Can make several programs work together
- Disadvantage
  - Little support for large and complicated programming semantics
  - Shell scripts need to be interpreted hence are slower programs

echo - to print to stdout

- The echo command can be used in a shell script to print text, to the terminal display
- Syntax: echo <zero or more values>
- Examples:

```
echo "Hello World"
echo "hello" "world" #two values
echo hello #need not always use quotes
echo "please enter your name"
```

Shell Environment Variables

- These are variables provided as part of the shell's operational
- They exist at startup but can be changed
- Examples are: USER, HOME, PATH, SHELL, HOSTNAME
- The "setenv" command (in tcsh) is used to set these, for example, by:
  - setenv PATH $PATH:/home/here/bin(this sets the PATH variable so that it's current value isappended by :/home/here/bin)
  - Note that setenv is how tcsh sets the environment variables

User-defined variables

- You can also specify variables yourself and these can also be used inside a script
- In tcsh, the "set" command is used to set a variable to a string value
- Form: `set<name>=<value>`
- Examples:

```
set firstVar = "any string"
set secondVar = 3
set mypath = /home/special/public_html
```

User-defined variables

- Once a variable has been defined, it's value can be used by "dereferencing" it with $

```
ls -al $mypath
```

- Note that using setenv or set without any parameters simply displays the current settings

Shell variables

- Note that for all shells, variables need not be declared explicitly, but simply used
- For the Bourne shell, the use is as follows (note that there should be no blanks before and after the equals sign and no need for the set command.
- Form: <name>=<value>
- Examples:

```
firstVar="hello world"
secondVar=45
echo $firstVar $secondVar "third argument"
```

- Note that $firstVar is the value of the variable firstVar

Shell Arguments

- Arguments on the command line can be passed to a shell script, just as you can pass command line arguments to a program
- $1, $2, ..., $9 are used to refer to up to nine command line arguments (similar to C's argv[1], argv[2], ..., argv[9]).
- Note that $0 contains the name of the script (argv[0])
- Example

```
shprog.sh john 40
shprog.sh bob 45 "new york"
```

Shell Arguments - Example

- Script

```
#!/bin/sh
#script name is greeting.sh
#display of today's date after a greeting
echo "Hello" $1 $2 ", pleased to meet you"
echo "The date is"
date
exit
```

- Usage:

```
>greeting.sh Spongebob Squarepants
```

Shell Arguments - Example

- $# contains the number of command line arguments.
- $@ will be replaced by a string containing the command line arguments.
- Example script: `echoArgs.sh`

```
#!/bin/sh
echo "The" $# "arguments entered:" $@
```

- Usage `echoArgs.sh Val1 Val2 Val3`
- Output: `The 3 arguments entered:  Val1 Val2 Val3`