Vi Editor and Shell Programming

by Mohamad H. Danesh

Agenda

- Some Useful command
  - Overview of previous session
  - Some extra commands
- Linux Job Control
- File Permissions
- Using VI Editor
- Basic Shell Programming

Some Useful Commands

Bash

- History for each user
- Use arrow keys
- !! meant the entire previous command.
- !$ meant just the last word of the previous command
- !<start_of_command> last command started with ...
- CTRL-R: search history
- history: see command history
- #!/bin/bash

pushd/popd/dir

- You can push the current directory onto a stack using the pushd command.
- Then you can switch to any directory you wish.

- If you wish to get back to the directory that you had earlier pushed onto the stack all you have to do is use the popd command.

- cd /tmp
- pushd /etc
- popd

finding-things utilities

- basic Linux finding-things utilities:
  - locate, find, and grep.
  - locate and find search for files, and grep searches for text in files.
  - Locate is the easiest. It maintains its own database of files on your system, so it's blazing fast.

finding-things utilities (cont.)

- find <path> <regular expression>
  - name, size, time, type, permission, ...
    - find /etc -name samba
- locate hello.c
- grep -ri "permitroot" /etc

Regular Expressions

- Powerful language for specifying strings of text to be searched and/or manipulated.

- Used by
  - grep “Get Regular Expression and Print” – search files line by line
  - sed Simple Editing tool, right from the command line
  - awk Scripting language, executes “program” on matching lines
  - perl Pathological Rubbish Lister. Powerful programming language

Regular Expressions: Summary

- Fundamentals:

.	Match *any* character(except EOL)
[character class]	Match the characters in *character class.*
[^character class]	Match anything *except* the character class.
$	Match the end of the line
^	Match the beginning of the line
*	Match the preceeding expression zero or more times
?	Match the preceeding zero or one time
|	Match the left hand side OR the right side
(regexp)	Group the regular expression
\	Treat next character literally (not specially)

- Examples:

Match a line beginning with a space-padded line number and colon.

```
^[ \t]*[0-9][0-9]*:
```

Match my name (various spellings)

```
(Tim Shelling)|(TJS)|(T\. Shelling)|(Timothy J\. Shelling)
```

Match if the line ends in a vowel or a number:

```
[0-9aeiou]$
```

Match if the line begins with anything but a vowel or a number:

```
^[^0-9aeiou]
```

Escaping and Quoting

Escaping: Providing a \ (backslash) before the wild-card to remove its special meaning.

- ✓ Escaping the *
- ✓ Escaping the [ ]
- ✓ Escaping the Space
- ✓ Escaping the \
- ✓ Escaping the newline character

Quoting: Enclosing the wild-card, or even the entire pattern, within quotes. Anything within these quotes (barring a few exceptions) are left alone by the shell and not interpreted.

Job Control

Jobs and processes

- **Job control** is a feature provided by many shells (including bash and tcsh) that let you control multiple running commands, or **jobs**, at once.
- Every time you run a program, you start what is called a process.
  - The command ps displays a list of currently running processes
- A running process is also called a job.
  - The terms process and job are interchangeable.

Foreground and background

- Jobs can either be in the **foreground** or in the **background**.
  - Foreground:
    - only be one job in the foreground at a time
    - foreground job is the job with which you interact
  - Background
    - jobs in the background do not receive input from the terminal
  - The shell assigns a job number to every running job

Unix job control

- Start a background process:
  - > python
    Hit CTRL-Z
    bg
- Where did it go?
  - jobs
  - ps
- Terminate the job: kill it
  - kill %*jobid*
  - kill *pid*
- Bring it back into the foreground
  - fg 1
- top : display top CPU processes, sort the tasks by CPU usage, memory usage and runtime

File Ownership and Permissions

File Ownership and Permissions

- Every file in UNIX has an owner, a group, and a set of permissions.
- You can use the ls command to view the permissons set for a file:
  - ls -l
  - Trwxrwxrwx n owner group size date name
- The first field, T:
  - _ for an ordinary file
  - d for an directory
  - **l** Symbolic link
  - p FIFO special file

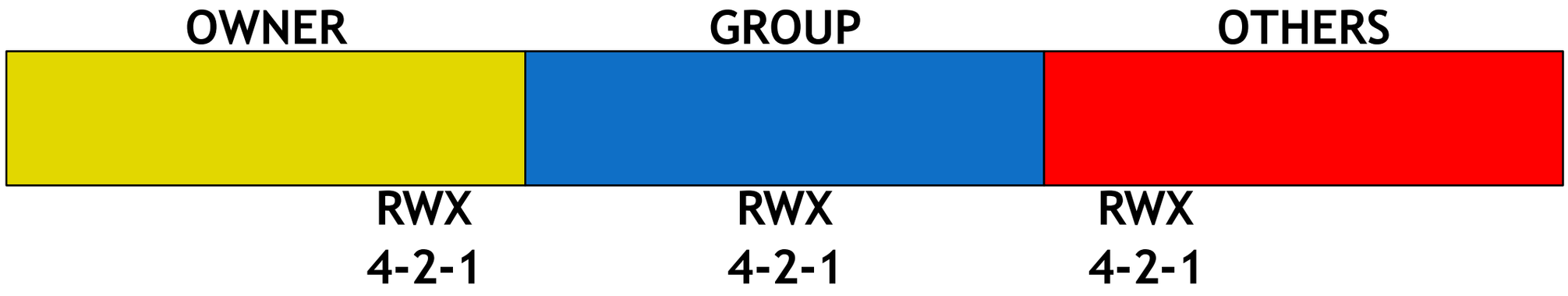
permissions cont'd

- Trwxrwxrwx n owner group size date name
- 3 sets of rwx represent the read, write and execute permission flags for the owner, the group, and others, reprectively.
- n represents the number of links to this file or directory
- owner represents the current owner of this file
- group represents the group associated with this file
- size is the number of bytes in this file
- date consists of date and time when the file was last modified
- name is the name of the file

Ls command, a bit more

- Colors meaning
  - global default normal file
  - directory
  - symbolic link
  - executable
  - compressed
  - media files
  - audio files
- Understand or change coloring:
  - # dircolors -p
  - /etc/DIR_COLORS

Change Permission, chmod



Chang permissions

chmod [ugo][+-=][rwx/binary] file

Example:chmod 754 myfile

- OWNER: Read Write and Execute
- GROUP: Read and Execute
- OTHERS: Read

Editors

What is vi?

- The visual editor on the Unix.
- Before vi the primary editor used on Unix was the line editor
  - User was able to see/edit only one line of the text at a time
- The vi editor is not a text formatter (like MS Word, Word Perfect, etc.)
  - you cannot set margins
  - center headings
  - Etc...

Editors

- emacs
  - Old and very user friendly
  - Menu based
- mcedit
  - A part of the midnight commander
  - Menu based, easy to use
- vi & vim (vi improved)
  - Difficult
  - Editor for programmers
  - Minimalist interface, Very little info displayed
  - Powerful shortcuts and commands

Vim equals Vi

- The current iteration of **vi** for Linux is called **vim**
  - **Vi** I**m**proved
  - http://www.vim.org

Starting vi

- Type **vi** <filename> at the shell prompt

- After pressing enter the command prompt disappears and you see tilde(~) characters on all the lines

- These tilde characters indicate that the line is blank

Vi modes

- There are two modes in vi
  - Command mode
  - Input mode

- When you start vi by default it is in command mode

- You enter the input mode through various commands

- You exit the input mode by pressing the Esc key to get back to the command mode

How to exit from vi
(comand mode)

- :q <enter> is to exit, if you have not made any changes to the file
- :q! <enter> is the forced quit, it will discard the changes and quit
- :wq <enter> is for save and Exit
- :x <enter> is same as above command
- The ! Character forces over writes, etc. :wq!

Moving Around

- You can move around only when you are in the command mode
- Arrow keys usually works(but may not)
- The standard keys for moving cursor are:
  - `h` - for left
  - `l` - for right
  - `j` - for down
  - `k` - for up

Moving Around

- `w` - to move one word forward
- `b` - to move one word backward
- `$` - takes you to the end of line
- <enter> takes the cursor the the beginning of next line

Moving Around

- `f` - (find) is used to move cursor to a particular character on the current line
  - For example, `fa` moves the cursor from the current position to next occurrence of '`a`'

- `F` - finds in the reverse direction

Moving Around

-) - moves cursor to the next sentence
- } - move the cursor to the beginning of next paragraph
- (- moves the cursor backward to the beginning of the current sentence
- { - moves the cursor backward to the beginning of the current paragraph

Moving Around

- Control-d scrolls the screen down (half screen)
- Control-u scrolls the screen up (half screen)
- Control-f scrolls the screen forward (full screen)
- Control-b scrolls the screen backward (full screen).

Entering text

- To enter the text in vi you should first switch to input mode
  - To switch to input mode there are several different commands
  - `a` - Append mode places the insertion point after the current character
  - `i` - Insert mode places the insertion point before the current character

Entering text

- I - places the insertion point at the beginning of current line
- o - is for open mode and places the insertion point after the current line
- O - places the insertion point before the current line

- R - starts the replace(overwrite) mode

Editing text

- `x` - deletes the current character
- `d` - is the delete command but pressing only d will not delete anything you need to press a second key
  - `dw` - deletes to end of word
  - `dd` - deletes the current line
  - `d0` - deletes to beginning of line
- There are many more keys to be used with delete command

Structure of vi command

- The vi commands can be used followed by a number such as
  n<command key(s)>
  - For example dd deletes a line 5dd will delete five lines.

- This applies to almost all vi commands

Undo and repeat command

- `u` - undo the changes made by editing commands
- `.` (dot or period) repeats the last edit command
- `^R`- Redo

Copy, cut and paste in vi

- `yy` - (yank) copy current line to buffer
- `nyy` - Where `n` is number of lines
- `p` - Paste the yanked lines from buffer to the line below
- `P` - Paste the yanked lines from buffer to the line above

(the paste commands will also work after the `dd` or `ndd` command)

Shell Script

What is a Shell Script?

- A Text File
- With Instructions
- Executable
- Why shell script?
  - Simply and quickly initiate a complex series of tasks or a repetitive procedure

What is a Shell Script?

```
% vim hello.sh
#!/bin/sh
echo ‘Hello, world’
% chmod +x hello.sh
% ./hello.sh
Hello, world
```

Notes:

Make a comment #

Parameters and Variables

- A shell parameter is associated with a value that is accessible to the user.
  - Shell variables
    - Names consist of letters, digits and underscores
      - By convention, environment variables use UPPERCASE
    - User created variables (create and assign value)
    - Keyword shell variables
      - Have special meaning to the shell
      - Being created and initialized by the startup file
  - Positional parameters
    - Allow you to access command line arguments
  - Special parameters
    - Such as
      - The name of last command
      - The status of most recently executed command
      - The number of command-line arguments

Shell Variables

Environmental variables are used to provide information to the programs you use. You can have both global environment and local shell variables.

- Global environment variables are set by your login shell and new programs and shells inherit the environment of their parent shell.
- Local shell variables are used only by that shell and are not passed on to other processes. A child process cannot pass a variable back to its parent process.

Some global environment variables are,

HOME	Path to your home directory
HOST	The hostname of your system
LOGNAME	The name you login with
PATH	Paths to be searched for commands
SHELL	The login shell you're using
PWD	Present working directory

Positional Parameters

- The command name and arguments are the positional parameters.
  - Because you can reference them 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
    - And so on up to $9
    - The rest of arguments have to be shifted to be able to use $1- $9 parameters.

Positional Parameters

- $1-$9 allows you to access 10 arguments
  - How to access others?
- Promote command-line arguments: 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
  - Makes additional arguments available
  - Repeatedly using shift is a convenient way to loop over all the command-line arguments

Positional Parameters

```
$ more demo_shift
#!/bin/bash
echo $1 $2 $3
shift
echo $1 $2
shift
echo $1
$ ./demo_shift 1 2 3
1 2 3
2 3
3
```

Special Parameters

- The number of arguments: $#
  - Return a decimal number
  - Use the test to perform logical test on this number
- 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
- Value of Command-line arguments: $* and $@
  - $* and $@represent all the command_line arguments (not just the first nine)
  - “$*” : treats the entire list of arguments as a single argument
  - “$@” : produce a list of separate arguments (Only bash/ksh/sh)

Shell scripts (contd.)

- The **expr** command:
  - Calculates the value of an expression.
  - E.g:
    - count=0
    - echo $count
    - count=`expr $count + 1`
    - echo $count

Notes on *expr*

- Why do we need the **expr** command ???
- E.g:

```
[axgopala@nitrogen public]$ file=1+2
[axgopala@nitrogen public]$ echo $file
1+2
[axgopala@nitrogen public]$
```

NOTE: 1+2 is copied as it is into *val* and not the result of the expression, to get the result, we need expr.

Notes on *expr*

- expr supports the following operators:
  - arithmetic operators: +,-,*,/,%
  - comparison operators: <, <=, ==, !=, >=, >
  - boolean/logical operators: &, |,!
  - parentheses: (,)
  - precedence is the same as C, Java

Control statements

- The three most common types of control statements:
  - conditionals: if/then/else, case, ...
  - loop statements: while, for, until, do, ...
  - branch statements: subroutine calls (good programming practice), goto (usage not recommended).

for loops

- for loops allow the repetition of a command for a specific set of values.
- Syntax:

```
for var in value1 value2 ...
do
   command_set
done
```

  - command_set is executed with each value of var (value1, value2, ...) in sequence

Notes on *for*

- Example: Listing all files in a directory.

```
#! /bin/bash

for i in *
do
  echo $i
done
```

NOTE: * is a wild card that stands for all files in the current directory, and *for* will go through each value in *, which is all the files and $i has the filename.

Conditionals

- Conditionals are used to “test” something.
  - In Java or C, they test whether a Boolean variable is true or false.
  - In a Bourne shell script, the only thing you can test is whether or not a command is “successful”.

Conditionals

- Every well behaved command returns back a return code.
  - 0 if it was successful
  - Non-zero if it was unsuccessful (actually 1..255)
  - This is different from C.

The *if* command

- Simple form:

```
if decision_command_1
then
    command_set_1
fi
```

Example

```
if [ $1 -gt 100 ]
then
  echo Hey that\'s a large number.
  pwd
fi
```

Using *elif* with *if*

```
if [ $n -lt 0 ]
then
    echo "Its Negative!"
elif [ $n -eq 0 ]
then
    echo "Its Neither Positive Nor Negative!!"
else
    echo "Its Positive!"
fi
```

Using *colon* in shell scripts

- Sometimes, we do not want a statement to do anything.
- In that case, use a colon ‘:’
  - if grep UNIX myfile > /dev/null
  - then
  - :
  - fi
  - Does not do anything when UNIX is found in myfile .

The *test* command

- Use for checking validity.
- Three kinds:
  - Check on files.
  - Check on strings.
  - Check on integers

Notes on *test*

- Testing on files.
  - test -f file: does file exist and is not a directory?
  - test -d file: does file exist and is a directory?
  - test -x file: does file exist and is executable?
  - test -s file: does file exist and is longer than 0 bytes?

Example – count executables in directory

```
# !/bin/bash
count=0
for i in *; do
if test –x $i
then
  count=`expr $count + 1`
fi
done
echo Total of $count files executable
```

NOTE: expr $count + 1 serves the purpose of count++

Notes on *test*

- Testing on strings.
  - test -z string: is string of length 0?
  - test string1 = string2: does string1 equal string2?
  - test string1 != string2: not equal?

Example

```
#! /bin/bash
if test -z $REMOTEHOST
then
:
else
  DISPLAY="$REMOTEHOST:0“
  export DISPLAY
fi
```

NOTE: This example tests to see if the value of REMOTEHOST is a string of length > 0 or not, and then sets the DISPLAY to the appropriate value.

Notes on *test*

- Testing on integers.
  - test int1 -eq int2: is int1 equal to int2 ?
  - test int1 -ne int2: is int1 not equal to int2 ?
  - test int1 -lt int2: is int1 less than to int2 ?
  - test int1 -gt int2: is int1 greater than to int2 ?
  - test int1 -le int2: is int1 less than or equal to int2 ?
  - test int1 -ge int2: is int1 greater than or equal to int2 ?

Example

```
# !/bin/bash
smallest=10000
for i in 5 8 19 8 7 3
do
if test $i -lt $smallest
then
  smallest=$i
fi
done
echo $smallest
```

NOTE: This program calculates the smallest among the numbers 5, 8, 19, 8, 3.

Notes on *test*

```
# !/bin/bash
smallest=10000
for i in 5 8 19 8 7 3
do
if [ $i -lt $smallest ]
then
  smallest=$i
fi
done
echo $smallest
```

- The test command has an alias ‘[]’.
  - Each bracket must be surrounded by spaces

While loop- Example

```
#! /bin/bash
i=1
sum=0
while [ $i -le 100 ]
do
  sum=`expr $sum + $i`
  i=`expr $i + 1`
done
echo The sum is $sum.
```

NOTE: The value of i is tested in the while to see if it is less than or equal to 100.

The *until* loop- Example

```
#! /bin/bash
x=1
until [ $x -gt 3 ]
do
  echo x = $x
  x=`expr $x + 1`
done
```

NOTE: The value of x is tested in the until to see if it is greater than 3.

Homework

- Read about /proc files
- Write a shell script which can show process information , including
  - Process ID
  - Command
  - Process Owner
  - status
  - CPU Usage
  - priority
- Deliverables
  - Brief Documentation about /proc files
  - Detail Documentation about files which used in homework
  - Well- defined shell script (good commenting)