CST8177 – Linux II

Review of Fundamentals

Todd Kelley
kelleyt@algonquincollege.com

Topics

- GPL
- the shell
- SSH (secure shell)
- the Course Linux Server
- RTFM
- vi
- general shell review

these notes

- These notes are available on Blackboard in PDF and txt format (text is good for grepping)
- Setting up a course website is on my to-do list

Linux Licensing: GPL (for example)

- You should be aware that we all use GNU and Linux (and other Free software) *under license*
- Q: who cares?   A: your employer
- When you receive a copy of GPL software, you are automatically granted a license from the copyright holder, and you have obligations
- Roughly, If you don’t give copies to others, no worries
- Roughly, If you give copies to others
  - 1. You must give the source code along with binary; OR
  - 2. You must provide a written offer to provide source code; OR
  - 3. for other special conditions or possibilities, read the GPL

Again, who cares?

- When you get a job, it will be incredibly important that your employer (through your work for them) is not found to be out of compliance with the GPL
- It gets serious when you (on behalf of your employer through their facilities) provide copies of software to others because you may inadvertently deny those others some rights
- Do not consider this legal advice: when/if the time comes, consult your employer's legal department
- "We always considered Open Source software to be a free-for-all under all circumstances. Why didn't anyone warn us?" -- I just did. That is all this was for.

The shell

- http://teaching.idallen.com/cst8207/12f/notes/120_shell_basics.html
- The shell is a program that we control by typing ascii characters for it to read
- Normally we are asking the shell to run programs for us on certain arguments, which we also type
- This is the *command line*
- Basically, the process is this
  - 1. The shell prints a prompt on our terminal or terminal emulator screen (“screen”)
  - 2. We type a command and “enter” (“return”)
  - 3. The shell carries out the operation in the way we asked (we might see output from that operation on our screen)
  - 4. repeat at step 1.

Sub-shells

- When you invoke `bash`, or `su`, or `sudo -s` you begin talking to a *sub-shell*
- schematically for illustration: see subshells.mp4 video

Sub-shells (cont'd)

- closer to what we actually see :

```
$ bash
bash
$ su
password:
# exit
exit
$ exit
exit
$
```

SSH and the Course Linux Server(CLS)

- all the details: http://teaching.idallen.com/cst8207/12f/notes/130_course_linux_server.html
- SSH (secure shell) is a program that allows us to securely invoke a shell *on a remote computer*
- On Windows: putty.exe
- schematically (abbreviated): see subshells.mp4 video

SSH and CLS (cont'd)

- what we'd see locally (abbreviated)

local terminal window

```
$ ssh user@cst8177.idallen.ca
password:
CLS $ exit
$
```

SSH to CLS

- (demo CLS login from Unix terminal window)
- (demo CLS login with PuTTY.exe on Windows)

- cst8177-alg.idallen.ca represents an *internal* IP address that works only on campus: when on campus, use this one

- cst8177.idallen.ca must be used when off campus

- login id is your algonquin userid
- password is given verbally by your Prof(s) or another student

Editing a text file

- Text Editors
  - Windows
    - notepad, wordpad (gui required)
  - Unix
    - vi (vim), emacs, nano, pico
    - gedit (gui required – no good for CLS)
- You need to be able to edit text files without a GUI
  - start the editor
  - move around
  - make a change
  - save and quit
- vi: http://teaching.idallen.com/cst8207/12f/notes/300_vi_text_editor.html
- long into the CLS and issue the command, vimtutor

Looking things up

- It’s normal for Unix users of all kinds (novice to expert) to consult the manual (man pages) often.
- man man
- man –k <search term>

- It’s a required skill to be able to find information in technical documentation, (“grepping through documents”)
  - example: CST8207 course notes covers fundamentals

- Here’s the normal process when you encounter a concept that you don’t know or it’s become vague or you’ve forgotten
  - search for the term in the manual
  - search for the term in the course notes
  - search for the term on the web (be careful)
  - ask your professor

Commands, programs, scripts, etc.

Command

A directive to the shell typed at the prompt. It could be a utility, a program, a built-in, or a shell script.

Program

A file containing a sequence of executable instructions. Note that it's not always a binary file but can be text (that is, a script).

Script

A file containing a sequence of text statements to be processed by an interpreter like bash**, Perl, etc.**

Every program or script has a stdin**,** stdout**, and** stderr **by default, but they may not always be used.**

Filter

A program that takes its input from stdin **and send its output to** stdout**. It is often used to transform a stream of data through a series of pipes.**

Scripts are often written as filters.

Utility

A program/script or set of programs/scripts that provides a service to a user. (ls, grep, sort, uniq, many many more)

Built-in

A command that is built into the shell. That is, it is not a program or script as defined above. It also does not require a new process, unlike those above.

History

A list of previous shell commands that can be recalled, edited if desired, and re-executed.

Token

The smallest unit of parsing; often a word delimited by white space (blanks or spaces, tabs and newlines) or other punctuation (quotes and other special characters).

stdin

The standard input file; the keyboard; the file at offset 0 in the file table.

stdout

The standard output file; the terminal screen; offset 1 in the file table.

stderr

The standard error file; usually the terminal screen; offset 2 in the file table.

Standard I/O **(Numbered 0, 1, and 2, in order)**

stdin**,** stdout**, and** stderr

Pipe

Connects the stdout **of one program to the** stdin **of the next; the "|" (pipe, or vertical bar) symbol.**

A command line that involves this is called a pipeline

Redirect

To use a shell service that replaces stdin**,** stdout**, or** stderr **with a regular named file.**

Process

http://teaching.idallen.com/cst8207/12f/notes/770_processes_and_jobs.html

- **A process is what a script or program is called while it's being executed. Some processes (called daemons) never end, as they provide a service to many users, such as crontab services from** crond**.**
- **Other processes are run by you, the user, from commands you enter at the prompt. These usually run in the foreground, using the screen and keyboard for their standard I/O. You can run them in the background instead, if you wish.**
- **Each process has a PID (or pid, the process identifier), and a parent process with its own pid, known to the child as a ppid (parent pid). You can look at the running processes with the** ps **command or examine the family relationships with** pstree**.**

Child process

- **Every process is a child process, with the sole exception of process number 1 – the** init **process.**
- **A child process is forked or spawned from a parent by means of a system call to the kernel services.**
- **Forking produces an <u>exact</u> copy of the process, so it is then replaced by an exec system call.**
- **The forked copy also includes the environment variables and the file table of the parent.**
- **This becomes very useful when redirecting standard I/O, since a child can redirect its own I/O without affecting its parent.**
- **Each non-builtin command is run as a child of your shell (builtins are part of the shell process: man builtin).**

Parsing the command line

1. **Substitute** history
2. **Tokenize command line (break it into "words" based on spaces and other delimiters)**
3. **Update** history
4. **Process quotes**
5. **Substitute** aliases **and defined functions**
6. **Set up redirection, pipes, and background processes**
7. **Substitute variables**
8. **Substitute commands**
9. **Substitute filenames (called "globbing")**
10. **Execute (run) the resulting program**

History

- **The command history is a list of all the previous commands you have executed in this session with this copy of the shell. It's usually set to some large number of entries, often 1000 or more.**
- **Use** echo $HISTSIZE **to see your maximum entries**
- **You can reach back into this history and pull out a command to edit (if you wish) and re-execute.**

Some history examples

- **To list the history:**

  System prompt> history | less
- **To repeat the last command entered:**

  System prompt> !!
- **To repeat the last ls command:**

  System prompt> !ls
- **To repeat the command from prompt number 3:**

  System prompt> !3
- **To scroll up and down the list:**

  Use arrow keys
- **To edit the command:**

  Scroll to the command and edit in place

Redirection

- **Three file descriptors are open and available immediately upon shell startup:** stdin**,** stdout**,** stderr
- **These can be overridden by various redirection operators**
- **Following is a list of most of these operators (there are a few others that we will not often use; see** man bash **for details)**
- **Multiple redirection operators are processed from left to right:** redir-1 redir-2 **may not be the same as** redir-2 redir-1
- **If no number is present with** > **or** <**,** 0 **(**stdin**) is assumed for** < **and** 1 **(**stdout**) for** >**; to work with** 2 **(**stderr**) it must be specified, like** 2>

Operator	Behaviour
	Individual streams
< <u>filename</u>	**Redirects** stdin **from** <u>filename</u>
> <u>filename</u>	**Redirects** stdout **to** <u>filename</u>
>> <u>filename</u>	**Appends** stdout **onto** <u>filename</u>
2> <u>filename</u>	**Redirects** stderr **to** <u>filename</u>
2>> <u>filename</u>	**Appends** stderr **onto** <u>filename</u>
	Combined streams
&> <u>filename</u>	**Redirects both** stdout **and** stderr **to** <u>filename</u>
>& <u>filename</u>	**Same as** &>**, but do not use**
&>> <u>filename</u>	**Appends both** stdout **and** stderr **onto** <u>filename</u>
>>& <u>filename</u>	**Not valid; produces an error**

Operator	Behaviour
	Merged streams
2>&1	**Redirects** stderr **to the same place as** stdout**, which, if redirected, must already be redirected**
1>&2	**Redirects** stdout **to the same place as** stderr**, which, if redirected, must already be redirected**
	Special stdin processing ("here" files), **mainly for use within scripts**
<< <u>string</u>	**Read** stdin **using** <u>string</u> **as the end-of-file indicator**
<<- <u>string</u>	**Same as <<, but remove leading** TAB **characters**
<<< <u>string</u>	**Read** <u>string</u> **into** stdin

Command aliases

- **To create an alias (no spaces after alias name)**

```
alias ll="ls -l"
```

- **To list all aliases**

```
alias   or    alias | less
```

- **To delete an alias**

```
unalias ll
```

- **Command aliases are normally placed in your** ~/.bashrc **file (first, <u>make a back-up copy</u>; then use** vi **to edit the file)**
- **If you need something more complex than a simple alias (they have no arguments or options), then write a bash function script.**

Filename Globbing and other Metacharacters

Metacharacter	Behaviour
\	**Escape; use next char literally**
&	**Run process in the background**
;	**Separate multiple commands**
$xxx	**Substitute variable** xxx
?	**Match any single character**
*	**Match zero or more characters**
[abc]	**Match any one char from list**
[!abc]	**Match any one char** not **in list**
(cmd)	**Run command in a subshell**
{cmd}	**Run in the current shell**

Simple Quoting

- **No quoting:**

```
System Prompt$ echo $SHELL
/bin/bash
```

- **Double quote:** "

```
System Prompt$ echo "$SHELL"
/bin/bash
```

- **Single quote:** '

```
System Prompt$ echo '$SHELL'
$SHELL
```

Observations:

Double quotes allow variable substitution;

Single quotes do not allow for substitution.

Escape and Quoting

- **Escape alone:**

```
Prompt$ echo \$SHELL
$SHELL
```

- **Escape inside double quotes:**

```
Prompt$ echo "\$SHELL"
$SHELL
```

- **Escape inside single quotes:**

```
Prompt$ echo '\$SHELL'
\$SHELL
```

Observations:

Escape leaves the next char unchanged;

Double quotes obey escape (process it);

Single quotes don't process it (treat literally)

Filespecs and Quoting

```
System Prompt$ ls
a  b  c
System Prompt$ echo *
a b c
System Prompt$ echo "*"
*
System Prompt$ echo '*'
*
System Prompt$ echo \*
*
```

Observation:

Everything prevents file globs

Backquotes and Quoting

```
System Prompt$ echo $(ls)   # alternate
a b c
System Prompt$ echo `ls`        #   forms
a b c
System Prompt$ echo "`ls`"
a
b
c
System Prompt$ echo '`ls`'
`ls`
```

<u>Observations:</u>

Single quotes prevent command processing

Summary so far

Double quotes allow variable substitution

"$SHELL" **becomes** /bin/bash

Single quotes do not allow for substitution

'$SHELL' **becomes** $SHELL

Escape leaves the next char unchanged

\$SHELL **becomes** $SHELL

Double quotes obey escape (process it);

"\$SHELL" **becomes** $SHELL

Single quotes don't process it (treat it literally)

'\$SHELL' **becomes** \$SHELL

Everything prevents file globs

"*" '*' * **each become** *

Single quotes prevent command processing

'\`ls\`' **becomes** \`ls\`

Escaping quotes

```
System Prompt$ echo ab"cd
> "
abcd
System Prompt$ echo ab\"cd
ab"cd
System Prompt$ echo 'ab\"cd'
ab\"cd
System Prompt$ echo "ab"cd"
> "
abcd
```

More quote escapes

```
System Prompt$ echo "ab\"cd"
ab"cd
System Prompt$ echo don't
> '
dont
System Prompt$ echo don\'t
don't
System Prompt$ echo "don't"
don't
System Prompt$ echo 'don't'
> '
dont
```

Observations

Unbalanced quotes cause a continuation prompt

Unescaped quotes are removed (but their meaning is applied)

"hello" **becomes** hello

"$HOME" **becomes** /home/username

Quoting protects quotes, as does \ escaping

"don't" **and** don\'t **are the same, and OK**

Single quotes are more restrictive than double

```
System Prompt$ echo '$USER' "$USER"
$USER someusername
```

Warm-up for next lecture:

Add a Unix command

- create a simple shell script
- make it executable
- copy it to a directory in $PATH
- presto, we’ve extended UNIX