Introduction to Operating Systems: Lecture on UNIX-like shells

Daniel Bosk[1]

Department of Information Technology and Media (ITM),
Mid Sweden University, Sundsvall.

shell.tex 560 2013-01-03 14:48:32Z danbos

Overview

1. UNIX shells
   - Input, output and error streams
   - Redirections
   - Pipelines
   - Environment variables
2. Some useful tools
   - Regular expressions (regex)
3. Some shell examples
   - A shell example courtesy of McIlroy
   - How to transfer a set of files
   - How to create a simple VoIP system
4. Shell scripting
   - Execution flow-control constructs
   - Some example shell scripts

UNIX shells

- The shell interprets commands from the user and executes them.
- The UNIX design of the shell is to implement all commands as separate programs – *each of which does one thing and does that thing well*.
- These programs are located in /bin, /sbin, /usr/bin, etc.
- Shells are also programs, standard shells are located in /bin.
- The simplistic design of UNIX makes many different shells available, e.g.
  - Korn Shell, ksh(1),
  - Bourne Shell, sh(1),
  - Bourne Again Shell, bash(1), and
  - the X window system (X11), Xorg(1).

UNIX shells

Manual pages

- Access manual pages (man-pages) by using the command man(1).
- Usage: `man [section] name`
- The section is given within parentheses directly after the command name, e.g. man(1) or sh(1).

```
$ man 1 sh
[man-page of sh]
$ man sh
[man-page of sh]
$
```

- A man-page with the same name can occur in different sections, e.g. printf(1) and printf(3).

UNIX shells

apropos(1)

apropos(1) can be used to search in man-page titles and descriptions.

```
$ apropos print
cat (1)              - concatenate files and
   print on the standard output
lp (1)               - print files
lp (4)               - line printer devices
lpq (1)              - show printer queue status
lpr (1)              - print files
lprm (1)             - cancel print jobs
lpstat (1)           - print cups status
   information
printf (1)           - format and print data
printf (3)           - formatted output
   conversion
whoami (1)           - print effective userid
```

Input, output and error streams

- Three special (and always open) files (streams):

stdin	input from e.g. terminal (i.e. keyboard).
stdout	output from process to e.g. to terminal (i.e. display).
stderr	error messages are written to stderr.

- Both stdout and stderr are output streams usually displayed in the terminal.
- Occationally these three streams are referred to by numbers, their file descriptors:

stdin	filedescriptor 0
stdout	filedescriptor 1
stderr	filedescriptor 2

Redirections

- > redirects output.

```
$ echo testing to redirect some output >
    /tmp/test.txt
$ cat /tmp/test.txt
testing to redirect some output
$
```

- < redirects input.

```
$ cat
test 1 2 3
test 1 2 3
$ cat < /tmp/test.txt
testing to redirect some output
$
```

Pipelines

- The pipe: redirects stdout of one process to stdin of another.

```
$ echo test 1 2 3 | cat
test 1 2 3
$ ls / | cat
bin
etc
usr
[...]
$
```

Environment variables

- Can be accessed by all processes.
- Can be used to store settings for some tools and utilities.
  - PAGER path to user's preferred pager.
  - EDITOR path to user's preferred editor.
  - VISUAL path to user's preferred visual editor.
  - PATH a colon separated list of paths to directories containing executable files (commands).
  - HOME the path to the user's home directory.
  - PS1 sets prompt of the shell, see e.g. sh(1).
- More can be read about this in the man-page for your shell and environ(7).

Variable substitution

- A variable is created and assigned by doing: `VARIABLE=value`.
- A variable can be referenced by its name prefixed with a dollar-sign ($), i.e. `$VARIABLE`.
- An alternative way is `${VARIABLE}`.
- The variable reference is substituted with the value of the variable.
- There are some special purpose variables:

  * expands to all positional parameters `$1 $2 $3 ....`

  # expands to the number of positional parameters.

  0 expands to the name of the shell or shell script.

Variable substitution, continued

```
$ export EDITOR=vim
$ export PATH=${HOME}/bin:${PATH}
$ echo $PATH
/home/danbos/bin:/usr/bin:/bin:[...]
$ $EDITOR /tmp/test.txt
[opens /tmp/test.txt for editing with vim]
$ EDITOR=emacs $EDITOR /tmp/test.txt
$ echo $0 ${0}
/bin/pdksh /bin/pdksh
$
```

Command substitution

- Output from commands can be substituted into environment variables.
- This is done using `$(<command> <arguments>)`.

```
$ username=$(whoami)
$ echo $username
danbos
$
$ old_time=$(date)
$ sleep 5
$ echo the time 5 seconds ago was $old_time
the time 5 seconds ago was Fri Apr 13
    06:56:57 CEST 2012
```

Some useful tools

echo(1)	display a line of text
test(1)	check file types and compare values
find(1)	search for files in a directory hierarchy
tr(1)	translate or delete characters
uniq(1)	report or omit repeated lines
sort(1)	sort lines of text files
wc(1)	print newline, word, and byte counts for each file
cut(1)	remove sections from each line of files
join(1)	join lines of two files on a common field
paste(1)	merge lines of files
xargs(1)	build and execute command lines from standard input
grep(1)	print lines matching a pattern
sed(1)	stream editor for filtering and transforming text

Regular expressions (regex)

- Is a pattern matching language, some details in regex(7).
- Both grep(1) and sed(1) uses regular expressions.
- Searching within man-pages are done using regex.

Regex, continued

- Ordinary characters are matched by themselves.
- There are special characters which must be escaped:

`{}[].*^$?()|`.

Regex, continued

Quantifiers

- Asterisk (*): 0 or more.
- Question mark (?): 0 or 1.
- Braces, `{n}`: exactly *n*.
- Braces, `{n,m}`: either $n, n+1, \ldots,$ or *m*.
- Braces, `{n,}`: *n* or more.

Regex, continued

Ranges

- Dot (.): any character.
- Parantheses, `(a|b)`: a or b.
- Square parantheses, `[abc]`: either a or b or c.
- Square parantheses, `[^,abc]`: not a nor b nor c.
- Square parantheses, `[a-z]`: one letter between a and z.

Regex examples

The assignment instruction said *hand in three files named kommandon.txt, inl.txt and svar_1.3.doc*. Quite straight forward, right?

These are the regular expressions I needed in my script:

- `[Kk]omm?andon?\.txt(\.txt|\.doc|\.docx)?`
- `[Ii]nl\.txt(\.txt)?`
- `.*([Ss]var|inlupp).*1.*\..*3.*\.(odt|doc|docx)*`

A shell example courtesy of McIlroy

```
$ cat long_text.txt | \
> tr -cs A-Za-z ’\n’ | \
> tr A-Z a-z | \
> sort | \
> uniq -c | \
> sort -k1,1nr -k2 | \
> head
[ten lines of output]
$
```

How to transfer a set of files

```
$ tar -zcf - /path/to/files | \
> ssh user@host.domain.tld tar -zxf -
$
```

How to create a simple VoIP system

From OpenBSD to OpenBSD:

```
$ cat /dev/audio | compress | \
> ssh user@host.domain.tld "uncompress >
   /dev/audio" &
$
```

From Ubuntu to Ubuntu:

```
$ arecord | gzip | ssh user@host.domain.tld
   "uncompress | aplay" &
$
```

Shell scripting

- Just shell commands, redirections, pipes etc. written to a file.
- Thanks to the UNIX design, there is no difference reading from stdin with stdin being the keyboard and stdin being redirected from a file.
- The file is hence read by the shell process and executed.
- Note that it opens the file separately, it does not redirect it to stdin – although this is fully possible.

Execution flow-control constructs

- if-then, elif-then, else
- for-do
- while-do
- case

These can be read about in the man-page of the shell, e.g. sh(1).

if-then, elif-then, else

```
$ VARIABLE=Yes
$ if [ "$VARIABLE" = "Yes" ]; then \
> echo "OK"; \
> elif [ "$VARIABLE" = "No" ]; then \
> echo "Sure thing"; \
> else
> echo "Huh?"
> fi
OK
$
```

for-do

```
$ for i in 1 2 3; do \
> echo $i; \
> done
1
2
3
$
```

Some example shell scripts

libris a script which fetches book information based on ISBN [for source see Bos10].

rm a delayed remove command [for source see Bos12].

References

[Bos10] Daniel Bosk. libris: a script for fetching reference information, 2010.

[Bos12] Daniel Bosk. rm: a script to delay removal of files, 2012.