

Unix commandline tools



Unix Command-line tools

Susanne Hoche
21.11.2006

Unix – kernel and shell



- The Unix *kernel* refers to the lowest-level, or 'inner-most' component of the operating system
- The Unix *shell* (aka command line) provides the traditional user interface for the Unix operating system
- It is called a *shell* because it hides the details of the underlying operating system behind the *shell's* interface
- The Unix shell is unusual since it is in both an interactive *command language* and a *scripting programming language*

Unix commandline tools

Unix shell



- The term **shell** also refers to a particular program, namely the **Bourne shell**, which was used in early versions of Unix and became a standard
- Every Unix(-like) system has an equivalent of the **Bourne shell**, e.g. the more feature-rich shell **bash** (Bourne again shell), **Korn shell**, etc.
- You can find out what shell you are using with the command **printenv SHELL** (an environment variable...)

```
$ printenv SHELL
/bin/bash
```

Environment variables



store “general information”

- Define an EV and make it known to the system:

```
$ export AN_ENV_VAR=/usr/local/bin
```
- Get the *value* of an EV:

```
$ printenv AN_ENV_VAR
/usr/local/bin
```
- Get the value of all currently defined EVs:

```
$ printenv
USER=hoche
HOME=/home/staff/hoche/linux
PATH=/usr/local...
.....
```
- Use an EV (note the \$!)

```
$ cd $AN_ENV_VAR
$ pwd
/usr/local/bin
```

Unix commandline tools

Directories



- A UNIX system comes with an existing file system tree
- Directory names in a UNIX file or directory path are separated by a forward slash "/"
- When you log in, you start somewhere down in that tree, in your **HOME** directory

Examples

- / - the root directory
- /usr - directory usr (sub-directory of the root directory "/")
- /home/staff/hoche/linux - my HOME directory
- . - the current directory
- .. - the current directory's parent directory

Home Directory



- When you first login, your home directory will be your current working directory
- This is where your directory (sub)tree starts, and your files and subdirectories are saved
- You can find out where your HOME directory is using the command **printenv** on the environment variable **HOME**

```
$ printenv HOME
/home/staff/hoche/linux/
$ cd $HOME
$ pwd
/home/staff/hoche/linux/
$ cd $HOME      is equivalent to      $ cd
```

Unix commandline tools

Creating, deleting & moving around the file system



- **mkdir dir1 [dir2 ...]** creates directory **dir1** etc.
- **mkdir -p dir1/dir2** creates **dir2** (and **dir1** if it doesn't exist yet)
- **pwd** shows the current working directory
- **cd aDir** changes to directory **aDir**
- **cd \$AN_ENV_VAR** changes to the directory defined by the environment variable **AN_ENV_VAR**
- **rmdir dir1 [dir2 ...]** removes directory **dir1** etc. – *if empty*

Moving around the file system



- Change to the home directory & display current working directory:

```
$ cd $HOME  
$ pwd  
/home/staff/hoche/linux
```
- “Change” to the current working directory:

```
$ cd .  
$ pwd  
/home/staff/hoche/linux
```
- “Change” to the current working directory's parent directory (the one above the current directory in the directory hierarchy)

```
$ cd ..  
$ pwd  
/home/staff/hoche
```


Unix commandline tools

Changing file permissions: chmod



- **chmod 754 aFile** changes aFile's permissions to be
 - **rwX (read, write, and execute)** for the owner,
 - **rx** for the group, and
 - **r** for the world(7 = **rwX** = 111 binary,
5 = **r-X** = 101 binary ,
4 = **r--** = 100 binary)

Copying, renaming and deleting files



- **cp file1 file2** creates a copy of **file1** named **file2**
- **mv file1 file2** moves or renames **file1** to **file2**
(**CAUTION**: this overwrites **file2** without further notice!!)
- **rm file1 [file2 ...]** removes or deletes **file1** etc.
- **rm -r dir1 [dir2 ...]** recursively removes a directory and its entire contents (**CAUTION**: without further notice!!)

Unix commandline tools

I'm stuck... man, whatis, apropos

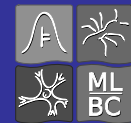


- **man** - formats and displays the on-line manual pages
 - **man aValidCommand**
 - **Navigation:** ENTER = one line down, SPACEBAR = page down, q=quit
- **whatis** - displays one-line command summaries
 - **whatis aValidCommand**

Finding what you want, given that you don't already know what you're looking for

- The **apropos** command looks through the headings of all the "man pages" and prints the title of any man page that matches the keyword
- E.g. **apropos string** to figure out how to e.g. search for a **string** in a file

Viewing and editing files



- By default, the following commands print to standard output (which goes to the terminal unless redirected somewhere else (...))
- **more** is a primitive filter for paging through text one screenful at a time
- **more aFileName** displays one page of **aFileName**'s contents at a time (navigation: ENTER = one line down, SPACEBAR = page down, q=quit)
- **less** provides extensive enhancements to **more**, provides better navigation (e.g. PageUp) , and doesn't - in contrast to **more** - have to initially read the entire file
- **less aFileName** displays one page of **aFileName**'s contents at a time

Unix commandline tools

Viewing and editing files



- **head aFile** displays the first 10 (if exist) lines of **aFile**
- **head aFile -n** displays the first **n** lines of **aFile**
- **tail aFile** displays the last 10 (if exist) lines of **aFile**
- **tail aFile -n** displays the last **n** lines of **aFile**
- **vi aFile** edits a file using the **vi** editor (which all UNIX systems will have in some form)
- **emacs aFile** edits a file using the **emacs** editor (not all UNIX systems will necessarily have **emacs** !)
- (**sed** is a stream editor used to perform basic text transformations on an input stream)
- **cat** and **paste** concatenate/merge files and display the result

Combining files: cat and paste



```
$whatis cat
cat (1) - concatenate files and print on stdout
$cat f* //meaning all files in the current directory starting with "f"
a
b
c
1
2
X
Y
Z

$whatis paste
paste (1) - merge lines of files
$paste f*
a      1      X
b      2      Y
c      Z
```

```
3 files:
//f1
a
b
c
//f2
1
2
//f3
X
Y
Z
```

Unix commandline tools

Searching for strings in files: grep



```
$whatis grep
grep - print lines matching a pattern
$more tmp1
cow
horse
pig
cow
$grep "o" tmp1
cow
horse
cow
$grep -c "o" tmp1
3
$grep -n "cow" tmp1
1:cow
4:cow
$grep -i "Cow" tmp1
cow
cow
```

Searching for strings in files: grep



```
$ grep ^c tmp1
cow
cow
$ grep e$ tmp1
horse
```

N.B.: ^ and \$ are metacharacters that respectively match the empty string at the beginning and end of a line

Unix commandline tools

Searching for strings in files: grep



- A *bracket expression* is a list of characters enclosed by [and]. It matches any single character in that list.
- Within a bracket expression, a *range expression* consists of two characters separated by a hyphen. It matches any character that sorts between the two characters, e.g. [a-e] is equivalent to [abcde].
- The regular expression [a-e] matches any character between a and e (i.e., any line with a character between a and e is a match).

```
$ grep [a-e] tmp1
cow
horse
cow

$grep -v [a-e] tmp1 //-v:Invert the sense of matching
//select non-matching lines
pig
```

Comparing files: diff



```
$whatis diff
diff - find differences between two files
e.g.:
```

- `diff -brief file1 file2` report only whether file1 and file2 differ, not the details of the differences
- `diff -i file1 file2` consider upper- and lower case letters as equivalent when comparing file1 and file2

Unix commandline tools

Redirection: >, >>, < and |



- You can redirect the output of most commands to a file (i.e. redirect *stdout*) with the redirection directives `>` and `>>` (file appending operator)
- **grep aString inputFile > outputFile** redirects the result of the **grep** command to **outputFile**
- **grep sString inputFile >> outputFile** redirects the result of the **grep** command to *the end of* **outputFile**
- UNIX commands that need input will usually read *stdin* (which normally comes from your keyboard)
- The redirection directive `<` can be used to redirect *stdin* – which is handy for UNIX commands that can't open files directly (so this example isn't really a good one:)
- `grep -c "o" < tmp1`
- `grep -c "o" tmp1`

Pipes: |



- The pipe symbol `|` is used to direct the output of one command to the input of another
- **ls -l | more** takes the output of the long format directory list command **ls -l** and pipes it through the **more** command (in this case a very long list of files can be viewed a page at a time)
- **grep -i aString inputFile | more**
- **grep -i String1 inputFile | grep -n String2**
- **head -10 inputFile | tail -2** displays the 8th and 9th line of **inputFile**

Unix commandline tools

Searching for files: find



```
$whatis find
```

```
find - search for files in a directory hierarchy
```

- **find searchPath -name fileName -print 2>/dev/null**
- **find . -name a.txt -print** finds all the files named **a.txt** in the current directory or any of its subdirectories (starting here '.')
- **find / -name a.txt -print** finds all the files named **a.txt** anywhere on the system (starting at the root directory '/')

Stream Editor sed



- Almost like **vi**-editor
- Unlike **vi**, **sed** does not change the file it works on: it takes a *stream* of data from standard input or a file, transforms it, and passes it to standard output
- Unlike **vi**, **sed** commands are *global*, i.e. a **sed** command is applied to *all* lines in the given file

Unix commandline tools

Substitutions with sed



sed 's/fromPattern/toPattern/'

```
$ more tmp
```

```
1 2 3
```

```
0 4 0
```

```
5 6 0
```

- substitute *the 1st* occurrence of '0' in each line of file `tmp` with '1':

```
$ sed 's/0/1/' tmp
```

```
1 2 3
```

```
1 4 0
```

```
5 6 1
```

- substitute *each* occurrence of '0' in each line of file `tmp` with '1':

```
$ sed 's/0/1/g' tmp
```

```
1 2 3
```

```
1 4 1
```

```
5 6 1
```

Substitutions with sed



sed 's/fromPattern/toPattern/'

- Substitute the occurrence of '0' in each line of file `tmp` with '1' if and only if it is the 1st character in the line :

```
$ sed 's/^0/1/' tmp      equiv. sed 's/^0/1/g' tmp
```

```
1 2 3
```

```
1 4 0
```

```
5 6 0
```

- Substitute the occurrence of '0' in each line of file `tmp` with '1' if and only if it is the last character in the line :

```
$ sed 's/0$/1/' tmp    equiv. $ sed 's/0$/1/g' tmp
```

```
1 2 3
```

```
0 4 1
```

```
5 6 1
```

N.B.: ^ and \$ are metacharacters that respectively match the empty string at the beginning and end of a line

Unix commandline tools

Substitutions with sed



```
$ more tmp2
a b c d e
f g h i j

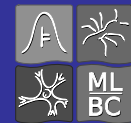
$ sed 's/^/predicate(/' tmp2
predicate(a b c d e
predicate(f g h i j

$ sed 's/ /,/g' tmp2
a,b,c,d,e,
f,g,h,i,j,
-----

$ sed 's/^/predicate(/' tmp2 | sed 's/ /,/g' | sed
's/$,/).' > outputfile

$ more outputfile
predicate(a,b,c,d,e).
predicate(f,g,h,i,j).
```

awk - pattern scanning and processing language



```
awk 'search pattern {programm actions}' inputFile
```

- **awk** searches `inputFile` for each line that contains search pattern, and performs for each found line the specified `programm actions`

```
$ paste f*
a      1      X
b      2      Y
c      3      Z
```

- print 1st and 3rd column of each line (empty search pattern)

```
$ paste f* | awk '{print$1,$3}'
a X
b Y
c Z
```

- print 1st and 3rd column of each line that contains an 'a'

```
$ paste f* | awk '/a/ {print$1,$3}'
a X
```

Unix commandline tools

Users



```
$whatis who
who - show who is logged on
$whatis whoami
whoami - print effective userid

$ who
eric      pts/0      Nov 11 12:46 (electra)
jewell    pts/1      Nov 21 12:51 (glo)
clark     pts/2      Nov 14 14:52 (freon)
xie       pts/3      Nov 15 12:05 (magma)

$ whoami
hoche
```

Processes



```
$whatis ps
ps - report process status
$ ps -ef | grep hoche
root 23921 2090 0 Nov23 ? 00:00:00 sshd: hoche [priv]
hoche  23923 23921 ...
hoche  23942 23923 ...
hoche  24407 23942 ... vim test
$ kill -9 24407
$ ps -ef | grep hoche
root 23921 2090 0 Nov23 ? 00:00:00 sshd: hoche [priv]
hoche  23923 23921 ...
hoche  23942 23923 ...
```

Unix commandline tools

History repeats itself



- Most shells include a powerful history mechanism that lets you recall and repeat past commands, potentially editing them before execution (especially useful when typing a long or complex command!)

- **history** displays the list of executed commands

```
$history
....
1014 cd /home/staff/hoche/linux/lintest/
1015 vi ex_sed-s.5
```

- You can use the up/down-arrow keys to access your previous commands and re-execute them
- Or, you can use “!” to select a specific command

```
$ !1014
cd /home/staff/hoche/linux/lintest/
```

Correct history



- You can use “!” to re-execute the last command of a specific kind (e.g. the last command starting with an ‘l’)

```
$history
...
1014 ls
1015 less tmp
```

```
$ !l
less tmp
```

(or, for the more cautious ones, in 2 steps:
(1st print the command to the command line)

```
$ !l:p
less tmp
```

(then, if we like it, execute the command)

```
$ !l
less tmp
```

Unix commandline tools

Correct history



- You can also correct and re-execute a command (attempt)
 - *The wrong attempt:*

```
$head -1 myflie
head: myflie: No such file or directory
```
 - *The correction and re-execution:*

```
$ ^li^il
head -1 myfile
```

sort - sort lines of text files



```
$ more tt1
3
4
1
6
$ sort -g tt1 //general-numeric-sort: compare according to
general numerical value
1
3
4
6
$ sort -gr tt1 //-r reverse order
6
4
3
1
```

Unix commandline tools

for – looping command



```
for ID in "183" "186" "192" "237" "1067"
do
  grep -v $ID $1 > $2
done
```

Command traces



```
UNIX: truss aCommand
$ truss man truss
Linux:
$ strace aCommand
```

Unix commandline tools

Manipulating Tape Archives: tar



- Tape archives are collections of files or directory trees
- One common use for **tar** is creating archive files: you can package several directories into an archive, you can archive directories that include links, you can preserve file ownership and access permissions, etc.
- To create a **tar** archive, use the **c** (create) and **f** (filename) options (and **v** to “verbosely” list files processed):
tar cvf archiveName file1 [file2 ...]
- To list the contents of a **tar** archive, use the **t** option:
tar tf archiveName
- To extract files from a **tar** archive, use the **x** option:
tar xvf archiveName file1 [file2 ...]
- add files to, delete files from a **tar** archive, ...

File compression / expansion



- **(un)compress filename**
- **gzip file, gunzip file**
- **bzip2, bunzip2 file: block sorting file compressor**

Unix commandline tools

Resources



- man pages
- <http://www.cs.bris.ac.uk/Teaching/Resources/General/Solaris/>
- <http://freeengineer.org/learnUNIXin10minutes.html>

Regular Expressions



- A regular expression describes a *sequence* of characters
- Regular expressions can be usefully combined with Unix commands such as `grep`, `sed`, etc.
- Regular expressions are case-sensitive; `A` does not match `a`
- There are two main types of regular expressions: *simple* and *extended* regular expressions. `sed`, `grep`, `more(, ...)` understand just simple *regular* expressions, whereas the utilities `awk` understand *extended* regular expressions.

Unix commandline tools

Regular Expressions



There are three important parts to a regular expression:

1. *Anchors* are used to specify the position of the pattern in relation to a line of text
2. *Character sets* match one or more characters in a single position
3. *Modifiers* specify how many times the previous character set is repeated

Regular Expressions: The Anchor Characters ^ and \$



- If you want to search for a pattern that is at the beginning or the end of a line of text, you use *anchors*
- The caret (^) is the starting anchor
- The dollar sign (\$) is the end anchor
- E.g., the regular expression ^A will match all lines that start with an uppercase A
- E.g., the expression A\$ will match all lines that end with uppercase A
- ^ is only an anchor if it is the first character in a regular expression; \$ is only an anchor if it is the last character

Unix commandline tools

Pattern Matches



- `^A` an A at the beginning of a line
- `A$` an A at the end of a line
- `A` an A anywhere on a line
- `$A` a \$A anywhere on a line
- `^^` an A ^ at the beginning of a line
- `\$$` a \$ at the end of a line

Matching a Character with a Character Set



- The simplest character set is a character.
- Character sets can be combined by placing them next to one another
- E.g., the regular expression **the** contains three character sets: **t**, **h**, and **e**. It will match any line that contains the string **the**, including the word **other**.

Unix commandline tools

Matching a specific character



- You can use square brackets `[]` to identify a specific character to be matched.
- The pattern that will match any line of text that contains exactly one digit is `^[0-9]$` (or, equivalently, `^[0123456789]$`, i.e, the hyphen specifies a character range)
- You can intermix explicit characters with character ranges
- E.g., the pattern `[A-Za-z0-9_]` will match
 - a single character that
 - is an upper- or lower-case letter,
 - digit, or
 - underscore

Matching a Character with a Character Set



- `T[a-z][aeiou]` will match a word that
 - is three letters long,
 - starts with an uppercase T,
 - has a lowercase second letter, and
 - has as third letter a lowercase vowel
- `^T[a-z][^aeiou]` will match a word that
 - is three letters long,
 - starts with an uppercase T,
 - has a lowercase second letter, and
 - has as third letter anything BUT a lowercase vowel

Unix commandline tools

Matching a Character with a Character Set



- You can also combine a character set with an anchor

```
$more tmp1
cow
horse
pig
cow
ducc
$ grep c tmp1
cow
cow
ducc
$ grep ^c tmp1
cow
cow
$ grep c$ tmp1
ducc
```

Regular Expressions



- A regular expression is not limited to literal characters.
- You can use *metacharacters* to expand or limit the possible matches of a regular expression
- There are also two basic types of metacharacters: (1) those that can be *evaluated to a single character*, and (2) those that *modify how characters that precede it are evaluated*

Unix commandline tools

Metacharacters (1): “.”



- The dot “.” metacharacter - can be used as a "wildcard" to match a *single* character
- By itself, “.” will match any character, except the end-of-line character
- E.g., we can specify the regular expression **A.E** and it will match **AAE, ABE, ACE, ...** (“.” will match any character in the position following A)
- E.g., the pattern that will match a line with any single character is: **^.\$**

Metacharacters (2): “*”



- The asterisk * metacharacter is used to match 0 or more *occurrences* of the *preceding* regular expression (which typically is a *single* character)
- Unlike * as a *shell* metacharacter (where it also means "zero or more characters"), the * by itself **does not match anything in a regular expression; it modifies the regular expression before it**
- E.g., the regular expression **.*** matches *any* number of characters (* modifies the preceding “.”)
- E.g., the regular expression **A.*E** matches *any* string that matches **A.E** - but it will also match *any* number of characters between A and E: e.g., **AIRPLANEE, A FINEE, AE, or A LONG WAY FROM HOMEE**

Unix commandline tools

Regular Expressions in combination with UNIX commands



- More substitutions with **sed**

```
$ more tmp2
a b c d e
f g h i j

$ sed 's/^/predicate(/' tmp2
predicate(a b c d e
predicate(f g h i j

$ sed 's/ /,/g' tmp2
a,b,c,d,e,
f,g,h,i,j,
-----
$ sed 's/^/predicate(/' tmp2 | sed 's/ /,/g' | sed 's/$/,/).' >
  outputfile
predicate(a,b,c,d,e).
predicate(f,g,h,i,j).
```