Text processing and command line

Операционни системи, ФМИ, 2021/2022

Piping commands together

- piping allows the stdout from one program (on the left of the pipe) to become the stdin of another (on the right of the pipe) (*"the Unix way"*)
- `ls -al | less`
- `cut -d: -f6 /etc/passwd|sort|uniq -c|sort -rn`
- redirection and piping can be combined
- usually used for feeding stderr into the pipeline along with stdout
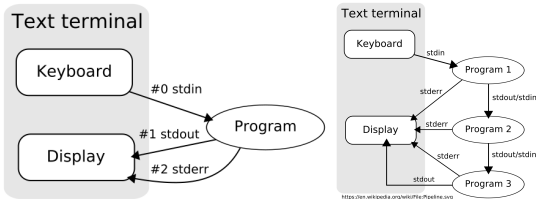- `ls /proc/ 2>&1 | grep kernel`

File (stream) redirection

- `sort < /etc/passwd`
- `echo 100000 > /proc/sys/fs/file-max`
- `ls -alR /proc/ 2> /dev/null`
- `ls -R /proc/ > output 2>&1`
    - `ls -R /proc/ &> output`

File (stream) redirection

- < stdin from a file
- > stdout to a file (overwrite)
- >> stdout to a file (append)
- 2> stderr to a file (overwrite)
- 2>> stderr to a file (append)
- &> both stdout and stderr

Recap on streams

- redirection - capturing output from a file, command, program, script, or even code block within a script and sending it as input to another file, command, program, or script
- pipe - passes the output (stdout) of a previous command to the input (stdin) of the next one

Combining files and merging text

- `cat` - concatenate files
- `paste` - merges text from multiple files
    - `-s` option to merge files serially
    - uses tabs as default delimiter

File statistics

- `wc` - print line, word, and byte counts for each file
  - `-c, --bytes` - print the byte counts
  - `-m, --chars` - print the character counts
  - `-l, --lines` - print the newline counts
  - `-w, --words` - print the word counts

Extracting columns of text

- `cut` - extracts sections from each line of files
  - `-c` - characters
  - `-f` - fields
  - `-d` - delimiter (TAB)
- most useful on structured input (text with columns)
- cannot change order ($N \leq M$)

Replacing text characters with `tr`

- translates one set of characters into another
  - `tr a-z A-Z`
- squeeze collapses duplicate characters
  - `tr -s '\n'`
- deletes a set of characters
  - `tr -d '\000'`

Text sorting

- `sort` sorts text
- can sort on different columns
  - `-k, --key=KEYDEF`
  - `-t, --field-separator=SEP`
- by default sorts in lexicographical order
  - 1, 2, 234, 265, 29, 3, 4, 5
- can be told to sort numerically (`-n`)
  - 1, 2, 3, 4, 5, 29, 234, 265
- can merge and sort multiple files simultaneously
- can sort in reverse order
- often used to prepare input for the `uniq` command

Duplicate removal utility

- `uniq` - removes duplicate adjacent lines from sorted text
- `-c` - prefixes each line of output with a number indicating number of occurrences
- ... then do numeric sort

Filename matching

- many commands take a list of filenames as arguments
- wildcard patterns
- historically called "file globbing"
- wildcard patterns are specified with special (meta) characters

Wildcard patterns

- ? - matches any single character
- * - matches anything (any number of characters)
- [...] - character classes
    - the – character denotes a range
    - examples: [abcd2345] [a-d2-5] [a-gA-Z0-5]

Brace expansion

- allows generation of arbitrary strings
- similar to wildcards, but target files or directories don't need to exist
- can have optional preamble and/or postamble
  - `{m,n,o,on}` expands to: m, n, o and on
  - `b{a,o,u,e,i}g` expands to: bag, bog, bug, beg, big
- can be combined with wildcards; brace expansion occurs before globbing

General quoting rules

- metacharacters \ ? ( ) $ ... * % { } [ ]
- backslash \
- double quotes " "
- single quotes ' '

Nesting commands

- command substitution - substitutes output of command in place of "embedded" command
- `command` - do *not* use
- $(command) - preferred method

Multiple and multi-line commands

- entering multiple commands on one command line
  - separate commands with a semi-colon ;
- entering multi-line commands
  - use backslash \
  - *line wrapping* / *continuation*

Searching inside files

- `grep` searches for patterns within files
  - `-n` shows line numbers
  - `-A NUM` prints match and NUM lines after match
  - `-B NUM` prints match and preceding NUM lines
  - `-C NUM` prints match and NUM lines before and after
  - `-i` performs case insensitive match
  - `-v` inverts match; prints what doesn't match
  - `--color` highlight matched string in color

Regular expressions

- Regular Expressions (REs) provide a mechanism to select specific strings from one or more lines of text
- complex language
- `grep, sed, perl, ...`
- `man 7 regex`

grep

- Global Regular Expressions Print
  - `grep := grep -G` (Basic RE)
    - originally, nondeterministic finite automaton (NFA)
  - `egrep := grep -E` (Extended RE)
    - originally, deterministic finite automaton (DFA)
  - `fgrep := grep -F` (fixed strings, not RE)
  - `rgrep := grep -r` (recursive)
  - `grep -P` (PCRE – Perl-Compatible Regular Expressions)
    - additional functionality
    - `pgrep` is unrelated
- difference between BRE and ERE depends on the implementation (i.e., GNU grep vs. others)

- most characters, letters and numbers match themselves
- special characters are matchable
- . matches any single character
- specify where the match must occur with anchors

RE special characters

- \t tab
- \n newline/line feed
- \r carriage return
- \f form feed
- \c control characters
- \x character in hex
- . any single character

RE anchors

- `^RE` anchor RE at start of line
- `RE$` anchor RE at end of line
- `\<RE` anchor RE at start of word
- `RE\>` anchor RE at end of word

RE character classes

- character classes, [...], match any single character in the list
  - sets - RE [0123456789] matches any single digit
- some predefined character classes
  - [:alnum:] [:alpha:] [:cntrl:] [:digit:]
  - [:lower:] [:punct:] [:space:] [:upper:]
- the – character denotes a range
- RE [[:alnum:]] equivalent to [0-9A-Za-z]
  - matches any single letter or number character

RE character classes examples

- `grep [[:upper:]] /etc/passwd`
- `egrep '^[rb]' /etc/passwd`
- `egrep '^[^rb]' /etc/passwd`

RE quantifiers

- control the number of times a preceding RE is allowed to match
- * match 0 or more times
- + match 1 or more times
- ? match 0 or 1 times
- {n} match exactly n times
- {n,} match at least n times
- {n,m} match at least n but not more than m times

RE quantifiers

```
egrep '^[stu].{14}$' /usr/share/dict/words
egrep '^[aeiou].{9}ion$' /usr/share/dict/words
egrep '^c.{15,}$' /usr/share/dict/words
egrep '^n.{6,10}c$' /usr/share/dict/words
```

RE parenthesis

- (RE) creating a new atom
  - `abc{3}` vs. `(abc){3}`
- (RE1|RE2) alternation: RE1 or RE2
  - `egrep '(dog|cat)' file`
- (RE)\n non-zero digit - storing values
  - `egrep --color '(.)\1' /etc/passwd`

The streaming editor

- sed stream editor for filtering and transforming text
- usually the output of another program
- often used to automate edits on many files quickly
- small and very efficient
- -i option for in place edits with modern versions

```
$ cat file
Parenthesis allow you to store matched
patterns.

$ sed -r 's/(.)\1/\[\1\1\]/g' file
Parenthesis a[ll]ow you to store matched
pa[tt]erns.
```

Text processing with awk

- `awk` – pattern scanning and processing language
- Turing complete programming language
- splits lines into fields (like `cut`) (`awk -F ':'`)
- regex pattern matching (like `grep`)
- math operations, control statements, variables, IO...
- *Aho, Alfred V., Brian W. Kernighan, and Peter J. Weinberger. The AWK programming language. Addison-Wesley Longman Publishing Co., Inc., 1987.*

Running an AWK program

- from the command line
  - `awk 'program code' input files`
  - `... | awk 'program code'`
  - single quotes
- from a separate file
  - `awk -f progfile.awk input files`
  - `... | awk -f progfile.awk`
  - as an awk script
    - first line: `#!/usr/bin/awk -f`
    - executable permissions
    - `... | ./foo.awk`

The structure of an AWK program

- each program is a sequence of one or more pattern-action statements

```
pattern { action }
pattern { action }
...
```

- data input is read line by line
- every input line is tested against each of the patterns in turn
- for each pattern that matches, the corresponding action is performed
  - action may involve multiple steps

AWK pattern-action statements

- single pattern-action statement
  - `$3 == 0 { print $1 }`
- no pattern
  - `{ print $1 }`
  - performed for every input line
- no action
  - `$3 == 0`
  - print each line that the pattern matches

AWK simple output

- print every line
  - `{ print }`
  - `{ print $0 }`
- print certain fields
  - `{ print $1, $3 }`
- number of fields
  - `{ print NF, $1, $NF }`
- number of lines read (so far)
  - `{ print NR, $0 }`
- computation
  - `{ print $1, $2 * $3 }`
- with text
  - `{ print "name:", $1, "calc:", $2 * $3 }`

AWK fancier output

- `printf(format, val-1, val-2, ..., val-n)`
  - format is verbatim text with % specifications
- `{ printf("%s has $%.2f\n", $1, $2 * $3) }`
- `{ printf("%-8s $%6.2f\n", $1, $2 * $3) }`

AWK selection

- comparison $2 >= 5
- computation $2 * $3 > 50
- text content $1 == "Susie"
- regular expressions /ar/
- combination of patterns && || !
  - $2 >= 4 || $3 >= 20
  - lines that satisfy both conditions are printed only once
  - different from two patterns:
    - $2 >= 4
      $3 >= 20
  - !($2 < 4 && $3 < 20)

AWK special patterns

- BEGIN
  - ```
    BEGIN { print "NAME    RATE    HOURS"; print "" }
          { print }
    ```
- END
  - ```
    $3 > 15 { emp = emp + 1 }
    END     { print emp }
    ```
  - ```
    END { print NR }
    ```
  - ```
            { sum += $2 * $3 }
      END { print "average", sum/NR }
    ```

AWK text

- variables can hold strings
  - `$2 > maxr { maxr = $2; maxemp = $1 }`
    `END       { print maxr, "for", maxemp }`
- string concatenation
  - `    { n = n $1 " " }`
    `END { print n }`
- printing the last input line
  - NR retains its value in an END action, $0 does not
  - `    { last = $0 }`
    `END { print last }`
- number of characters in a string
  - `{ print $1, length($1) }`

AWK++

- control-flow statements
  - `if-else`
  - `while`
  - `for`
- arrays
- examples
  - `awk -F ':' '$1 ~ "oo" { print $2 }'`
  - `awk '$1 != 1 { print $2 }'`
  - `awk -v "foo=${BAR}" '....'`
    - BAR is shell variable, copied as `foo` in AWK

Text editing

- Unix revolves around text
  - text is robust
  - text is universally understood
  - the only tool / program required is a text editor
  - remote administration possible over low-bandwidth connections
- Text editors
  - Many editors available, each with fanatical followings
  - `pico`/`nano`, `vi` and `emacs` are the most common
  - `$EDITOR` control default editor

vi / vim

- `vi` The Visual Editor
  - Developed originally by Bill Joy for BSD UNIX
  - Officially included in AT&T UNIX System V
  - Available on all UNIX platforms
- `vim` Vi IMproved
  - Has significantly enhanced functionality
  - Includes a compatibility mode

vi help

- Books & Cheat Sheets
- `:help`
- http://www.vim.org/
- `vimtutor`

Basic vi

- Insert Mode: keystrokes are inserted into the document
- Command Mode: keystrokes are interpreted as commands
- `hjkl`
- `i a [ESC] x dd`
- Saving & exiting
    - `:w`
    - `:q`
    - `:wq`
    - `:wq!`