Text processing and command line

Операционни системи, ФМИ, 2022/2023

special characters & quoting

special character

- has a meaning beyond its literal meaning, a meta-meaning
    - `;` – separate multiple commands on one command line
    - `\` – multi-line command (line wrapping)

quoting

- `\` – backslash
- `" "` – partial (weak) quoting[1]
- `' '` – full (strong) quoting

[1] does not interfere with variable substitution

command substitution

what

- reassigns the output of a command into another context
- extracts the stdout of a command
- can be used:
  - as arguments to another command
  - to set a variable
  - for generating the argument list in a `for` loop
- invokes a subshell[2]
- may result in word splitting[3]

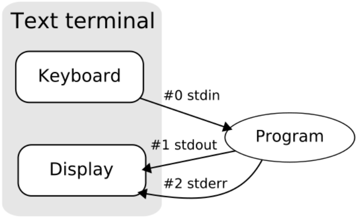
how

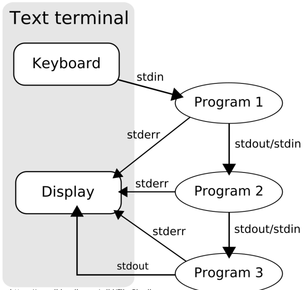
- `` `cmd` `` – classic form with backquotes (backticks); do *not* use
- `$(cmd)` – preferred form
  - `ls -l $(echo a.txt)`
  - permits easy nesting

[2]a child process launched by a shell (or shell script)

[3]dividing a character string into separate and discrete arguments

streams





https://en.wikipedia.org/wiki/File:Pipeline.svg

- *pipe* – passes the output (stdout) of a previous command to the input (stdin) of the next one
- *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

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`

notation	meaning
<	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

- redirection and piping can be combined
- usually used for feeding stderr into the pipeline along with stdout
  - `ls -R /proc/ 2>&1 | fgrep denied`

combining files and merging text

- `cat` – concatenate files and print on the standard output
- `paste` – merge lines of files
    - `-s, --serial` – paste one file at a time instead of in parallel
    - uses TABs as default delimiter

file statistics

- `wc` – print newline, 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 OPTION... [FILE]...`
  - print selected parts of lines from each FILE to standard output
  - `-c, --characters=LIST` – select only these characters
  - `-d, --delimiter=DELIM` – use DELIM instead of TAB for field delimiter
  - `-f, --fields=LIST` – select only these fields
  - `LIST` – one range, or many ranges separated by commas
  - `N`
  - `N-`
  - `N-M`
  - `-M`
  - same order ($N \leq M$)
- most useful on structured input (text with columns)

replacing text characters with `tr`

- `tr [OPTION]... SET1 [SET2]`
  - translates one set of characters into another
    - `tr a-z A-Z`
  - `-d, --delete` – delete characters in SET1, do not translate
    - `tr -d '\000'`
  - `-s, --squeeze-repeats` – collapse duplicate characters
    - `tr -s '\n'`

text sorting

- `sort [OPTION]... [FILE]...`
  - write sorted concatenation of all FILE(s) to standard output
  - `-r, --reverse` – reverse the result of comparisons
  - `-n, --numeric-sort` – compare according to string numerical value
- can sort on different columns
  - `-t, --field-separator=SEP` – use SEP instead of non-blank to blank transition
  - `-k, --key=KEYDEF` – sort via a key; KEYDEF gives location and type
  - `KEYDEF` is `F[.C][OPTS][,F[.C][OPTS]]`

`uniq(1)`, `comm(1)` & `join(1)`

- `uniq [OPTION]... [INPUT [OUTPUT]]`
  - filter adjacent matching lines from INPUT (or standard input)
  - matching lines are merged to the first occurrence
  - `-c, --count` – prefix lines by the number of occurrences
  - `-f, --skip-fields=N` – avoid comparing the first N fields
- `comm [OPTION]... FILE1 FILE2`
  - compare sorted files FILE1 and FILE2 line by line
  - `-1` – suppress column 1 (lines unique to FILE1)
  - `-2` – suppress column 2 (lines unique to FILE2)
  - `-3` – suppress column 3 (lines that appear in both files)
- `join [OPTION]... FILE1 FILE2`
  - join lines of two files on a common field
  - FILE1 and FILE2 must be sorted on the join fields

searching inside files

- `grep [OPTION...] PATTERNS [FILE...]`
  - searches for PATTERNS in each FILE
  - `-n` – prefix each line of output with the line number within its input file
  - `-A NUM` – print NUM lines of trailing context after matching lines
  - `-B NUM` – print NUM lines of leading context before matching lines
  - `-C NUM` – print NUM lines of output context
  - `-i, --ignore-case` – ignore case distinctions in patterns and input data
  - `-v, --invert-match` – invert the sense of matching, to select non-matching lines
  - `--color` – display matched strings in color

Regular Expressions

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

`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)

RE

- most characters, letters and numbers match themselves
- special characters are matchable
  - `\t` – tab
  - `\n` – newline/line feed
  - `\r` – carriage return
  - `\f` – form feed
  - `\c` – control characters
  - `\x` – character in hex
- `.` – matches any single character
- specify where the match must occur with 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 examples

```
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`

sed(1)

- stream editor for filtering and transforming text[4]
- usually the output of another program
- often used to automate edits on many files quickly
- small and very efficient
- `-f script-file` vs. `-e script`
- `-E, -r` – extended RE
- `-i[SUFFIX]` – edit files in place (modern versions)
- `s/regexp/replacement/`

```
$ 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.
```

[4] https://pubs.opengroup.org/onlinepubs/9699919799/utilities/sed.html

text processing with AWK

- `awk` – pattern scanning and processing language [5]
- Turing-complete programming language
- splits lines into fields (like `cut`) (`awk -F ':'`)
- regex pattern matching (like `grep`)
- math operations, control statements, variables, IO…

[5] 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 required is a text editor
  - remote administration possible over low-bandwidth connections
- text editors
  - many editors available, each with fanatical followings[6]
  - `pico`/`nano`, `vi` and `emacs` are the most common
  - `$EDITOR` – control default editor

[6] https://en.wikipedia.org/wiki/Editor_war

`vi` & friends

`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
- `neovim` – heavily refactored `vim` fork

`vi` help

- books & reference cards
- `: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`
  - `:q!`
  - `:wq`