Linux For Embedded Systems

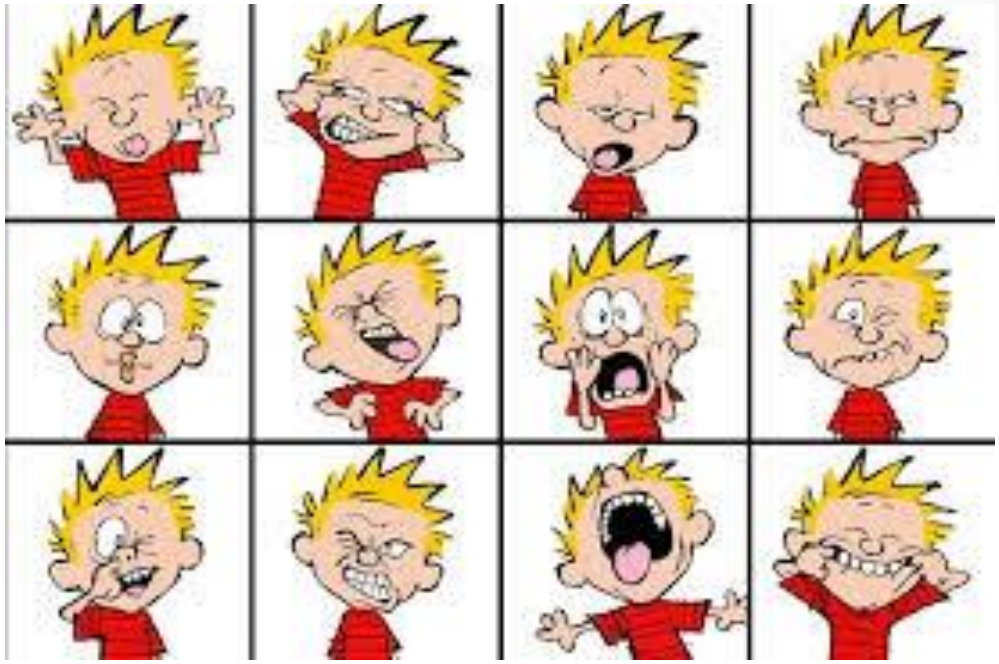
For Arabs

Course 102:

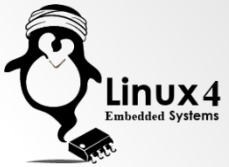
Understanding Linux

Ahmed ElArabawy

Lecture 13: Regular Expressions



What is Regular Expressions ??



- We learned how to do simple text operations (like search on single strings…)
- How about if I want to,

  Search for string-1 or string-2 …

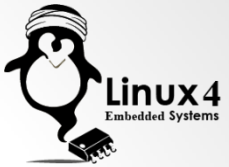
  Search for a string only if it occurs at the beginning of the line

  Search for a pattern (such as a phone number, email, URL,…)

  Search for a pattern that have some repetition

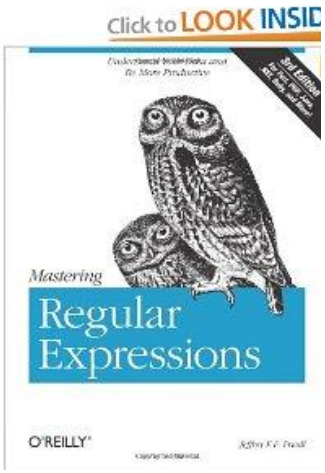
  etc...
- This means we need a more powerful mechanism to deal with text patterns….
- That is the role of Regular Expressions (REGEX)

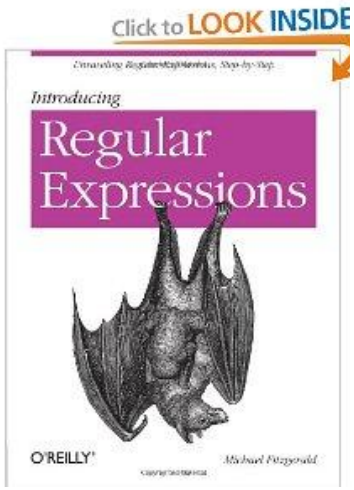
What is Regular Expressions ??

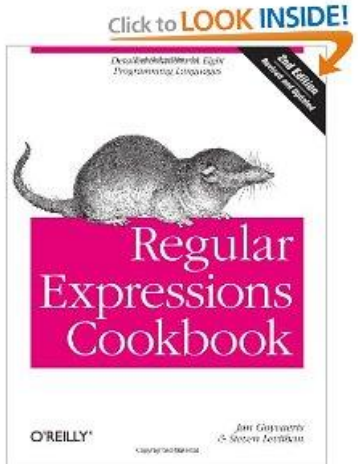


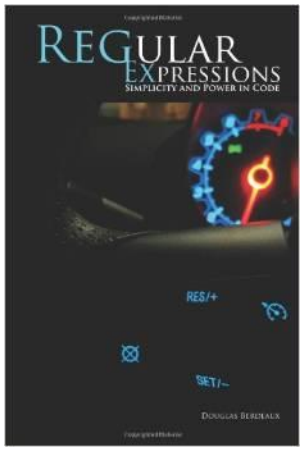
- We learned to use patterns with filenames via wild cards and other expansions but those expansions were limited to file names
- Regular expressions use similar techniques but used to general text patterns (so don't confuse the two, cause they have different rules)
- Regular Expressions are a very powerful tool for creating text patterns for use by several Linux tools (specially for search in text)
- Used a lot in scripting languages and tools such as '***perl***' and '***python***'
- One of the main tools using it is ***grep*** (the '***re***' in '***grep***' stands for regular expressions)

Regex Are Complicated









- A lot of Special Cases and forms
- Some of the tools using it, have slight changes to the rules
- Some tools have richer set than the one used in bash (such as **perl**)
- This lecture in only going to introduce the topic, if interested there are a lot of sources on REGEX

grep Command

$ grep [options] <string/pattern> <files>

Search for the string or pattern within the set of files

Option	Long	Description
-i	**--ignore-case**	Ignore case
-v	**--invert-match**	Show lines that does not match
-c	**--count**	Print only the count of the matches
-l	**--files-with-matches**	Print only the filenames
-L	**--files-without-match**	
-n	**--line-number**	Include line numbers
-h	**--no-filename**	Don’t include the file name

Literal Vs. Meta-Characters

- Literal characters are those characters that represent themselves in the search pattern

  $ grep "error" *.log

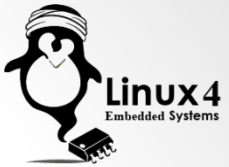
  The letters in "error" are all literal characters

- Meta characters are those characters that have special meaning,

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

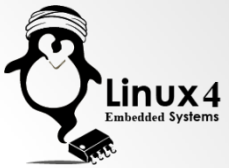
  All other characters are literal characters

Literal Vs. Meta-Characters



- Normally, it is recommended to put any expression with meta-characters inside a quote, to avoid expansion according the shell rules
- Meta characters can be treated as literals if they are ***escaped***, i.e. preceded by a back slash
  - Examples, **\^ \{ \$ **
- The back slash can also convert some of the literal characters into a meta-characters
  - Examples : **\d \w**
- The regular expressions are patterns of text created from a mix of literal characters and meta-characters

Types of Regular Expressions



- POSIX defines two types of regular expressions,
  - Basic Regular Expressions (BRE)
  - Extended Regular Expressions (ERE)

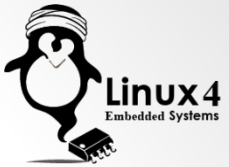
- Basic Regular Expressions use the following meta-characters, all other characters are considered literal:

  . ^ $ [ ] *

- Extended Regular Expressions use the following set in addition to the basic set,

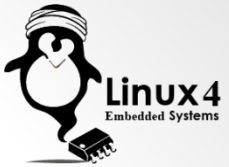
  ( ) { } ? + |

- Then the backslash is used to reverse those meta-characters into literals (in ERE), and vice versa (in BRE)
- The tool '***grep***' uses BRE
- To access ERE, use '***egrep***' or '***grep –E*** '



BASIC REGULAR EXPRESSIONS (BRE)

The “ANY” Character (The dot character ‘ . ’)

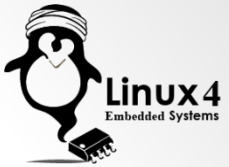


- The ‘**.**’ (dot) is a meta-character that represent any single character (not including NULL character)
- For example,

  $ grep “.zip” file.log

  This searches for a 4 letter text pattern that starts with any character followed by the three letters in ‘zip’
  - May result in: ***gzip***, ***bzip***, ***gnubzip*** , ***rezipped***
  - But will not result in : ***zip***

Anchor Characters (^) and ($)



- The (**^**) at the beginning of the string means that this string has to be at the beginning of the line
- The (**$**) at the end of the string means that this string has to be at the end of the line

- Examples:

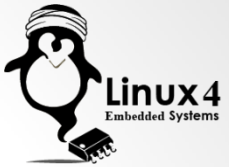
  $ grep "^zip" file.txt ➔ results in any line that starts with 'zip'

  $ grep "zip$" file.txt ➔ results in any line that ends with 'zip'

  $ grep "^zip$" file.txt ➔ results in any line that have only 'zip'

  $ grep "^$" file.txt ➔ results in empty lines

Bracket Expressions ([ ) and ( ])



- The use of brackets for any of a set of characters listed between the brackets

  $ grep "[bg]zip" dict.txt

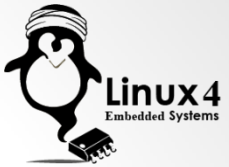
  Results in: **bzip**, **gzip**, **aabzip**
- Any character inside the bracket will be considered literal except for
  - (**^**) if it comes at the beginning (will considered as negation)
  - (**-**) if it comes in the middle (will be considered as range)
- Negation,

  $ grep "[^bg]zip" dict.txt
  - Will catch words with any character before zip except b or g or Null character
- Ranges

  $ grep "[a-z]2" dict.txt
  - Will catch words starting with any small letter followed by 2

  $ grep "[a-fA-F]4" dict.txt
  - Will catch any word with letter A-F (case insensitive) followed by 4

Shorthand Character Classes



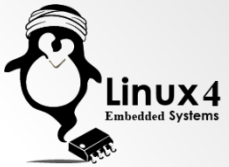
- We can use a selected set of escaped characters to represent some character classes
- Examples

regex	Description
\w	Stands for **[a-zA-Z0-9_]** (word character)
\s	Stands for space characters or tabs or line breaks
\t	Stands for tabs(ASCII 0x09)
\r	Stands for Carriage Return (ASCII 0x0D)
\n	Stands for Line Feed (0x0A)
\xnn	Stands for character with ASCII = nn (\xA9 == @)

Character Classes

- The following character classes can be used,

Class	Description
[[:alnum:]]	Alphanumeric **[a-zA-Z0-9]**
[[:word:]]	same as alnum with addition of underscore (**\w**) **[a-zA-Z0-9_]**
[[:alpha:]]	Only letters **[a-zA-Z]**
[[:digit:]]	Only Digits **[0-9]**
[[:blank:]]	Space Bar or Tab (**\s**)
[[:lower:]]	Only lower case letters **[a-z]**
[[:upper:]]	Only upper case letters **[A-Z]**
[[:space:]]	space
[[:xdigit:]]	Hex digit **[a-fA-F0-9]**



EXTENDED REGULAR EXPRESSIONS (ERE)

Alternation
(|)

$ grep -E "AAA|BBB" file.txt

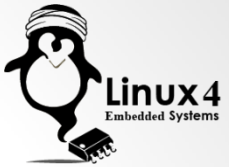
This matches any line containing AAA or BBB

- We separate the alternation from the rest of the regular expression using '**()**'

$ grep –E "^(AAA|BBB|CCC)" file.txt

This matches any line starting with AAA or BBB or CCC

Quantifiers (*,+, and ?)



- The character '**?**' is used to express that the preceding element to be optional (zero or one time)
- The character '*****' is used (zero or More times)
- The character '**+**' is used (One or More times)
- Example,

  We are searching any line starting with a phone number… this can be in the format

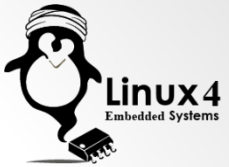
  (nnn) nnn – nnnn OR **nnn nnn-nnnn**

  $ grep –E "^\(?[0-9][0-9][0-9]\)? [0-9][0-9][0-9]-[0-9][0-9][0-9][0-9]" file.txt

- Example,
- Want to check which lines constitute a proper statement, start with a capital letter, followed by any character or spaces, and ending by a period

  $ grep –E "^[[:upper:]][[:upper:][:lower:]]*\.$" file.txt

Matching Count ({) and (})



- The curly brackets '**{}**' can be used to match an element specific number of times

Matching Count	Description
{n}	n times
{n,m}	n to m times (inclusive)
{n,}	n or more times
{,m}	m or less times

- Example: The phone number example:

$ egrep "\(?[:digit:]{3}\)?[:digit:]{3}-[:digit:]{4}" file.txt

Examples

- Some example to play with:

  $ egrep -i 'Continued\.*' file.txt

  $ grep "[0-9a-fxA-FX]abc" file.txt

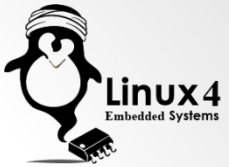
  $ egrep 'fish\|chips\|pies' file.txt

  $ egrep -i '\(cream\|fish\|birthday\|\) cakes' file.txt

  $ grep '[Jj]oe [Bb]loggs' file.txt

  $ grep -E "colou?r" file.txt

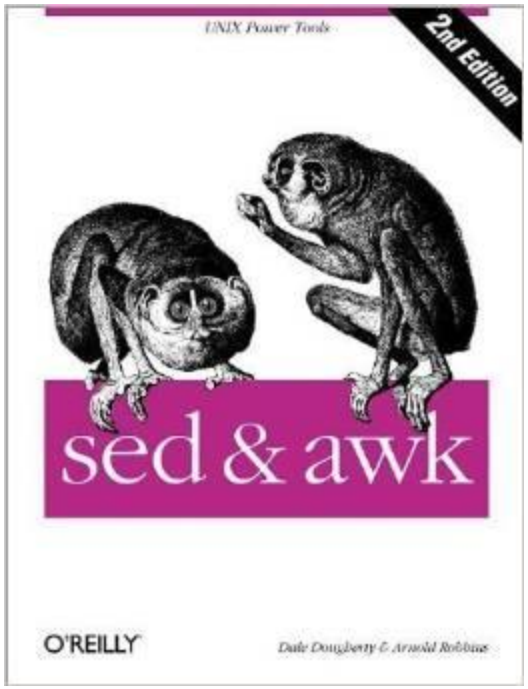
Using Regular Expressions in Other Commands



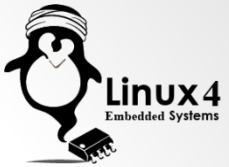
- A lot of other commands also work with regular expressions such as,
  - The commands '***find***' and '***locate***' for finding files
  - The '***vi***' editor
  - The command '***less***' can perform text search using regular expressions
  - Tools that use regular expressions extensively
    - ***sed***
    - ***awk***

Linux4 Embedded Systems

sed & **awk** TOOLS



sed



- ***sed*** stands for Stream Editor
- ***sed*** is used to modify text files within a program, script, or from the command line
- ***sed*** is one of the very famous programs to use regular expressions extensively

- ***sed*** is a rich program, and has many options, so we are just providing a quick introduction for it in this lecture
- We will discuss some of its famous patterns, and usage

Syntax

- All sed commands are formed into a command string that is passed to sed in the following format

  $ sed 'command string'

- The input file that sed works on can be passed through
  - Input redirection

    $ sed 'command string' < input-file

  - Pipes

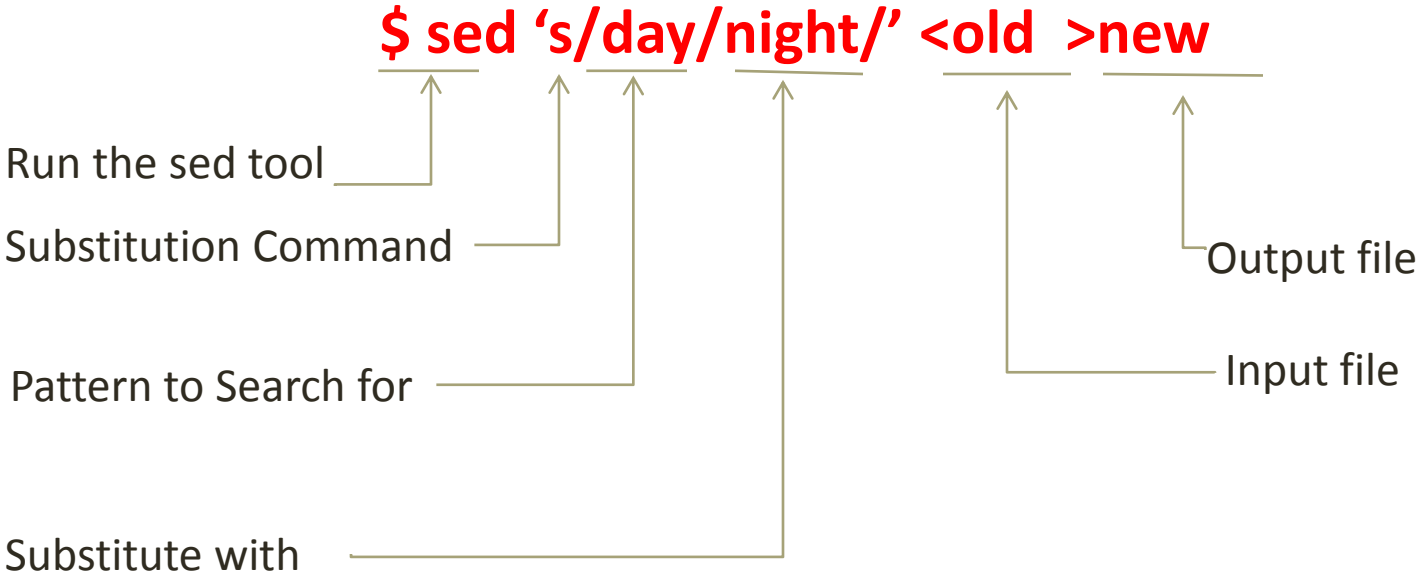
    $ cat input-file | sed 'command string'

- The output goes to stdout, in case you want the output in a file, you need to redirect it

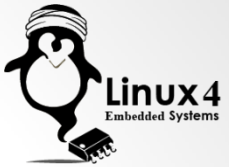
  $ sed 'command string' < input-file > output-file

Substituting Text (The ‘s’ Command)

- Substituting text is one of the most popular commands of sed
- It searches for all the occurrences of a certain text pattern (using REGEX) and substitute it with different string
- Example:



Selecting the delimiter Character



- By default, **sed** uses the slash as a separator (delimiter) between the command, search pattern, and the replacement pattern.
- If we need to use a slash inside the search pattern or the replacement pattern, we must escape it (precede it with a back slash)

  $ sed 's/yes\/no/true\/false/' < old-file > new-file

- Another option is to use another character as a separator,

  $ sed 's:yes/no:true/false:' < old-file > new-file

  $ sed 's|yes/no|true/false|' < old-file > new-file

Using the Global Option

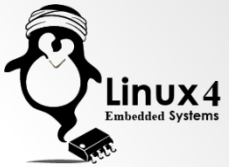
- **sed** works on the file line by line
- So this command

  $ sed ‘s/day/night/’ <old-file >new-file

  - Changes one occurrence of the search pattern (day) in each line into the replacement pattern (night)
  - If the search pattern exists multiple times inside the same line, only the first occurrence will be substituted
- To avoid that use the global flag

  $ sed ‘s/day/night/g’ <old-file >new-file

Using the Matched String (&)



- Suppose that we want to search for all occurrences of a string, and want to put it inside brackets. We would use the command

  $ sed 's/abc/(abc)/g' <old-file >new-file

- What if we want to use a pattern for our search , then we can use the meta-character & to indicate the matched string

  $ sed 's/gr[ae]y/(&)/g' <old-file >new-file

  $ sed -r 's/[0-9]+/(&)/g' <old-file >new-file

- Another example, let us say we want to repeat any number in the file

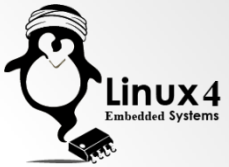
  $ sed -r 's/[0-9]+/& &/g' <old-file > new-file

Note, the last examples have been using '***-r***' to support Extended REGEX

awk

- awk is an excellent tool for processing files containing tabulated data (in rows and columns)
- awk also has strong string manipulation functions, so it can search for particular strings and modify the output
- awk also has support for associative arrays, which are very
- There is a GNU version of AWK called "gawk“
- The name “awk” comes from the names of its inventors

Syntax



- AWK reads the input file line by line
- On each line, it performs a test based on some pattern
- If the test succeeds it performs an action
- Thus an awk statement would look like:
  ***pattern* { action }**

- Some basic pattern,
  - No Pattern: perform the action on every line
  - ***BEGIN***: perform the action if this is the beginning of the file
  - ***END***: perform the action if this is the end of the file

- Example:
  BEGIN { print "Now we will start the file" }
  { print }
  END { print "That was the end of the file" }

More Advanced AWK scripts

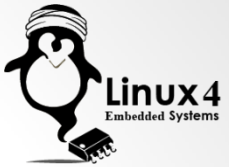
BEGIN { print "File\tOwner"}
{ print $9, "\t", $3}
END { print " - DONE -" }

- This will do the following ,
  - When the first line of the input file is read, the scripts outputs the strings “File” and “Owner” separated by a TAB
  - For every line read from the input file (including the first one, the data in the 9th column and that of the 3rd column are printed separated by a TAB
  - After the input file is fully read, the scripts prints “- DONE –”
- Now if you apply this awk statements on the output of

  $ ls -l

  It should make a list of file names and owner name for all files

More on AWK



- AWK has a very powerful syntax
- It can be written inside a bash script
- It can define variable, perform mathematical operations
- It also have a strong support for REGEX

Linux 4 Embedded Systems

http://Linux4EmbeddedSystems.com