Network Services

Unix Shell Scripts

Johann Oberleitner
SS 2006

Agenda

- Unix Command Line Processing
  - Filters
- Shell Scripts
- Regular Expressions
  - grep
- sed
- awk

UNIX Shells

- Shells are normal programs
  - Provides a command-line interface to OS
  - One shell is started after login
    - Which shell is stored in /etc/passwd
  - May be started from a shell
    - subshell
  - Link between end-user and operating system
- Supports execution of shell scripts
- Available on most operating systems

Shells

- sh - Bourne Shell
  - original shell
- bash - Bourne Again Shell
  - Advanced version of sh
- ksh – Korn Shell
  - Advanced version of ksh
- csh - C Shell
  - Some operations taken from C prog. Language
- tcsh – Tenex C Shell
  - Advanced version of csh
- Cmd.exe – WinNT-WinXP
  - Poor
- Powershell (MSH)
  - New Microsoft Shell
  - Many features as in UNIX shells

Bash

- Most used shell on Linux systems
- Available for most operating systems
  - also for Windows
- Feature rich
  - Compatible with sh
  - Most features as in ksh

Commands

command	options	Argument1...

- Command – name of command
- Option(s)
  - Modifies how command works
  - Usually Character(s) preceded by +/-
  - Sometimes no +/-
- Targets on which command operates

Builtin-Commands

- Provided by the shell itself
  - cd – change directory
  - pushd,popd – directory stack
  - fg,bg – job control commands
  - shift – shift command line arguments
  - exit (logout) – exit from (login) shell
  - ...

echo

- Copies input arguments to output
- Example:

```
$ echo simple test
simple test
```

man + help

- man
  - Manual pages for commands
  - man find
    - Shows manual page for the find command
- help
  - Help pages for built-in commands
  - help alias
    - Shows help page for the alias command

Commands for file system

- pwd – print working directory
- ls – list directory
- cd – change directory
- mkdir – make directory
- rmdir - remove directory

Commands for files

- cat – (con)catenates files
- more – prints file
  - If more than one page, waits on space key
- less similar – much better
  - Supports backward scrollingyx
- cp – copy files/directories
- mv – move files/directories
  - Also used for renaming
- rm – remove files/directories

Find files/directories

- find pathname criteria
- Finds all files in the directory (and subdirectories) given by pathname that satisfy the given criteria
- Example
  - find . –name abc
    - All files in local directory (and subdirectories) that have a name containing abc
  - find . –type f
    - Returns all files that are regular files (no directories, links, or other entities that are represented in the file system)

Shell Variables

- Variables have a *name*
- Can be referenced with $*name*

```
$ echo $SHELL
/bin/bash
    $SHELL is a predefined variable
```

- Variables are defined with =

```
$ x=abcdefg
echo $x
Abcdefg
```

- Variables are unset with unset

```
$ unset x
```

- All variables printed with set

Exit status

- On exit of a command a special variable is filled
- $?
  - Success: value is 0
  - Failer: value != 0

```
$ ls afilethatdoesnotexist; echo $?
1
```

Typed variables

- Declares typed variable with
  - declare option var1 ...
  - Option may be
    - -i integer

$ a=5; b=7
$ result=$a*$b
$ echo $result
5*7
	$ declare –i a=5 b=7
	$ declare –i result
	$ result=$a*$b
	$ echo $result
	35

Arithmetic Evaluation

- Bash supports arithmetic calcuations
- Evaluation via $((expression))
  - Variables may be defined as strings!
- Example

```
$ c=5
$ d=10
$ echo $((c+d*c+d))
58
```

Subshells

- Variables only defined in current shell
- When new shell is started variable is not known. Has to be exported.

$ x=abc
$ bash   starts subshell
$ echo $x
 	
(no output)
	$ x=abc
	$ export x
	$ bash
	$ echo $x
	abc

Standard Streams

- Commands take input and output from predefined standard streams
  - Some commands do not use this input
- Standard Input (stream desciptor 0)
- Standard Output (stream desciptor 1)
- Standard Error (stream desciptor 2)
- Streams may be redirected
  - Example: instead of keyboard a file may be used as input

Input Redirection

- Input redirection operator 0< (shorter: <)
  - Lets files be input source instead of keyboard
- Principle syntax:

```
command 0< inputfile
```

- Example
  - Files.txt contains a b

```
$ cat < files.txt
a b
```

Output redirection

- Output redirection operator 1>, 1>>, 1>| (shorter: >,>>,>|)
  - Redirects output to file instead of monitor
- Principle syntax:

  command 1> outputfile

  if file exists, outcome depends on noclobber option that forbids accidently destroying files by redirection,

  noclobber: $ set –o noclobber

  redirect to existing file leads to an error

  command 1>> outputfile (appends to file)

  command 1>| outputfile (always creates output file)
- Example
  - ls > filecontents.txt
- Error redirection
  - Via 2>, 2>>, 2>|

Output and Error redirection

- Redirecting to different files
  - ls 2>| error.txt 1>| output.txt
- If same file is used this may lead to an file already open error
  - >& has to be used
- Redirecting Output and Error to same file
  - ls 1> output.txt target 2>&1

Pipes

- Often output of one command needed as input of another command
- Instead of using files
  - Use | (=pipe) symbol
- Example (count files in a directory)
  - ls /etc > /tmp/etc_list   # copy dir to file
  - wc –l /tmp/etc_list   # wordcount files
- With Pipes:
  - ls /etc | ws -l

tee command

- Copies standard input
  - to standard output
  - AND to a file / multiple files

Multiple commands

- Sequence
  - Separated either by ;
  - In different lines
  - Example: echo abc; ls .
- Grouped
  - In round braces ()
  - Affects redirection
  - Example: (echo abc; ls .) > result.txt
- Conditional
  - Shell logical operators: && (=and) , || (=or)
  - Shortcut evaluation as in C/Java/C#
  - Example: cp nonExistingFile temp || echo "Copy failed"

Escape character

- Some characters have special meaning (metacharacters)
- Example:
  - <space> separates command parts
  - | <pipe> chains commands
  - $ initiates variable substitution
  - \, <, >, >>
- If character should be printed:
  - Escape with backslash \
  - Example: \$, \\, \|

Quotes

- Text in single quotes ' ' is removes meaning of metacharacters:
  - $ x='abc$ dfdf|xyz'; echo $x

```
abc$ dfdf|xyz
```

- Text in double quotes " " is similar
  - Except: dollar sign ($) keeps its meaning
    - Allows variable substitution in strings
  - $ y="begin $x end"; echo $y

```
begin abc$ dfdf|xyz end
```

Command substitution

- Execution of commands within strings
- $(command)
- In addition to variable substitution
- Example

```
echo "Das ist das heutige Datum: $(date)"
Das ist das heutige Datum: Thu Apr 27 ...
```

- Supports that (command) strings are built dynamically and executed via command substitution

Aliases

- Allows assigning a name to a command string
- alias aliasname=command
  - Has to put into quotes!
- Example: alias lhome="ls $HOME"
  - Lhome is a new command that lists all entries of the home directory (stored in the $HOME environment variable)
- Alias without arguments shows all defined aliases

Filter Commands

- Chaining different commands
- Most commands support input and output streams in text formats
- Filters support transformation of these text formats
- Chained via the pipe
- See Pipe & Filter Architectural Style
  - In software Architecture

Filter Commands

- cat – catenate
  - Concatenates files
- head – beginning of a file
- tail – end of a file
- cut – extracts columns
- paste – combines lines together
  - Columns of input files are put together for each row

Filter Commands

- sort - Sorts a file
  - Row-wise by fields as sort key
- uniq – deletes duplicate lines in sorted(!) files
- wc – count words,lines,characters
- diff – difference of two files
- Comm – commonalities among two files

Command-Line Processing / 1

- Processing Order of Commands

- 1. Split into tokens
- 2. Check if 1st token is opening token
  - Restart processing with nested command
- 3. Check if 1st token is alias
  - Substitute alias string instead of alias, restart

Command-Line Processing / 2

- 4. Brace expansion
  - Example: a{b,c} becomes ab ac
- 5. Tilde Expansion
  - ~ will be replaced with home directory
    - "ls ~" equivalent to "ls $HOME"
- 6. Perform variable substitution $name
- 7. Perform command substitution $(cmd)
- 8. Evaluate arithmetic expressions $((a+b))

Command-Line Processing / 3

- 9. Splits result into words
- 10. Pathname expansion (expand *, ? with files on disc)
  - Pathnames are substituted by shell
  - Unlike DOS or Windows shells
- 11. Uses first word as command
  - Searches command:
    1. Function in a script
    2. Built-in command
    3. File in any of the directories in $PATH
- 12. Setup redirection & start command

Shell Scripts

- Text file that contains shell commands
- Supports writing reuseable commands
- Shells provide constructs
  - Variables
  - Control flow (if,switch,loops)
  - Execution of commands

Shell Script Structure

- Interpreter Designator
  - First line of shell script
  - Example:
    - #!/bin/bash
  - On start of the shell designator is used to find correct shell interpreter for this script
- Shell commands
- Comments
  - Initiated with #
  - Shell designator is also comment

Execute Permissions

- Shell Scripts need Execute permissions
- Can be assigned with the chmod command
- Example:
- chmod o+x myscript
  - Gives owner of the file execut permissions
- chmod a+x myscript
  - Gives all users permission to execute script

A simple Script

```
#!/bin/bash
# first script
echo "A simple script"
ls /etc | wc
```

```
$ ./myScript
A simple script
        74        74        739
```

Conditionals

- For commands based on exit code
- Logical operators !, &&,|| supported
  - Executes commands, evaluation based on exit codes
- Condition tests
  - Condition within [ ] does not execute commands
  - String comparisons (=,!=,<,>,-n,-z)
    - -n tests string not null, -z tests string is null
  - File attribute checking
    - -a file exists
    - -d file exists and is a directory
    - -f file exists and is a regular file
  - Integer Conditionals
    - -lt, -le, -eq, -ge, -gt, -ne (less than, less than or equal, ...)

Conditional Constructs - Samples

1. ls filedoesnotExist
   - true if ls finds the file "filedoesnotExist"
2. [-a $filename]
   - True if a file with name $filename
3. [$s = "xyz"]
   - true if s contains the value xyz
4. [$i –eq 42]
   - true if i contains the integer value 42

Control Constructs / if

- Structure 1

```
if condition
then
    statements
fi
```

- Structure 2

```
if condition
then
    statements
else
    statements
fi
```

Control Constructs - Conditions

```
#!/bin/bash
if [ -a fileexists ]
then
   echo "fileexists exists"
else
   echo "fileexists does not exist"
fi
```

Parameters & Variables

- Variables identically used as on the command line
  - name=abc; echo $name
- Parameters
  - Can be provided on script startup
  - Referenced with $1,$2,$3,...
  - $0 is name of command
  - $# number of arguments
  - $* combines all arguments in one string
    - not possible to use arguments in calls to other commands
  - $@ list of all arguments
- Shift
  - Shifts command-line arguments left
  - shift 1 : 1=$2; 2=$3; 3=$4; ...

Control Constructs - loops

- While loop (as in Java)
  - Loops until condition becomes false

```
while condition do
   Statements
done
```

- Until loop
  - Loops until condition becomes true

```
until condition do
   Statements
done
```

Control constructs - loops

- for loop
- Lets you iterate over a fixed list of values

```
for varname in list
do
  statements that use $varname
done
```

for-loop Example

1.
```
for i in $@
do
   wc $i
done
```

2.
```
for i in $(ls /etc)
do
   wc "/etc/$i"
done
```

3.
```
numbers="1 2 3"
for i in $(echo $numbers)
do
   echo $i
done
```

Shell functions

- Functions within shell scripts
- Declared with "function name"
- Body inside curly braces {}
- Variables are global
- Local variables possible with local keyword

Shell functions example

```
#!/bin/bash

function myfunc
{
  echo "$# args"
}

myfunc "$*"

myfunc "$@"
```

Exit Status

- Return Code to Calling Shall
  - exit N
- exit 0
  - Command was ok
  - return code=0
- exit 1
  - Error code 1
- ...

Other constructs

- case
  - Similar to switch statement
- select
  - Provides a menu and waits for a selection
  - Like for loop
- Arithmetic for loop
  - Like for loop in C/Java/C#

Startup / Logoff scripts

- When user logs in
  - Login shell is started
  - Bash executes scripts from user's home directory
    - .bash_profile, .bash_login, .profile
      - Not normally shown because of . Prefix
      - Sets search path, terminal settings, environment variables
  - On ending login shell .bash_logout executed
    - cleanup
- When a bash subshell is started
  - executes .bash_rc from user's home directory

Regular Expressions

- Patterns of characters that are matched against text
- Used by grep, sed, awk to address target lines
- Atoms
  - Specify what text is to be matched and where it is found
- Operators

- Important to know which elements are supported in a tool

Atoms

- Single character
  - Must appear in the target text
- Dot (.)
  - Any character in the target text
- Class []
  - [ABC] or [A-Z] matches a class of characters
  - [^BC] characters not B or C
- Anchors
  - ^ beginning of line, $ end of line

Operators

- Sequence
  - Series of atoms, all atoms must be matched
- Alternation |
  - Either one or the other atom must be matched
- Repetition \{m,n\}
  - An atom must be matched from m to n times
  - NOT SUPPORTED by all tools!
- Short form *,+,?
  - * means zero or more times
  - + means one or more times
  - ? Means zero or one time
- Groupings ( )
  - Next operator after group applies to entire grouping

grep

- Name comes from a command in ed editor
  - Global regular expression print (g/re/p)
  - Variants:
    - egrep (extended grep),
    - fgrep (fast grep)
- Example:
  - egrep '^(e|fun)' *
    - Searches if lines exist that have either an e at the start of a line or a fun.

sed

- sed=Stream Editor
  - Not a real editor, no modification of input file
- Text Files
- Line-oriented
  - Each line of input file is scanned
  - Applies instructions to each line of a text file
  - Scripts may contain multiple instructions

sed - buffers

- "Pattern Space"
  - Buffer that sed uses for operations
  - Each input line is read and stored in the pattern space
- "Hold Space"
  - Additional buffer that is used for further operations
- Usually spaces work line-oriented
  - Larger amounts are supported
  - Must be constructed manually

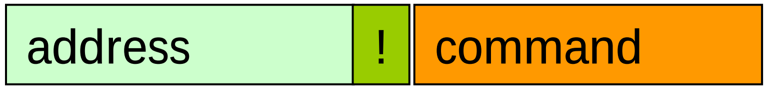
sed – working principle

```
foreach line in input file {
    copy line to "pattern space"
    foreach instruction in sed-script {
        if  instruction.address matches line
            apply instruction.command
    }
}
```

sed – options

- -n
  - No automatic output of pattern space
  - Allows scripts control of printing
- -e 'script'
  - Inline script (within calling command)
- -f scriptfilename
  - Invocation of file

sed – Script Format



- Address specifies which input lines shall be processed
- ! (optional) denotes if the address denotes denotes the complement (= if it denotes all lines that shall not be processed)
- Command specifies what shall be done with a line.
  Usually specified with a single character
  - Example p=print

Sed – Addresses

- Specifies which lines shall be processed
- 4 address types
    - Single-Line Address
    - Set-of-Line Addresses
    - Range Addresses
    - Nested Addresses

Sed – Single Line Address

- Matches one single line
  - Specified via line number
    - Eg. 377 denotes 377th line
  - Last line denoted via $
- Example
  - sed –n –e '2p'
    - Prints second line
  - sed –n –e '$p'
    - Print last line
  - sed –n –e '2!p'
    - Print all except second line

sed – Set-of-Line Addresses

- Matches each line that matches a regular expression
  - /regular expression/
- Example (sed command omitted):
  - '/Zeile/p' input.txt
    - Prints all lines that contain the string "Zeile"

sed – Range Addresses

- May match zero or more lines
  - start-address,end-address
    - Each address may be line-number
    - Each address may be a regular expression
- Example (sed command omitted)

2,4p	prints lines 2-4
/Das/,/Das/p	prints lines from first /Das/ to last.
1, /Das/p	prints lines from 1 to last with /Das/.

sed – Nested Addresses

- Address contained in another address
  - Nested address & command within { }
  - Command within nested address
- Example:
  - 1,3{

```
    /[E|e]ine/!p
  }
```

Prints all lines within the first three lines that contain neither the word 'Eine' nor 'eine'.

sed - Commands

- Modify Commands
  - insert (i) – inserts a text before address
  - append (a) – appends a text after address
  - change (c) – replaces line with text
  - delete (d) – deletes line
  - Substitute (s) – replaces text

sed – Modify Command Samples

```
#Insert text before first line
1i\
 /*\
    * Class: \
    * Task:\
    * Creation Date: 22.02.2006\
    ...
 */
```

```
sed –f creationsig.sed MyClass.java
```

sed - substitute

address	s	/regexp/newtext/	flag

- Deletes text matched by regexp
- Instead uses newtext
- Flags:
  - 1,2,3,... replacement of n-th occurence of regexp
  - g = global replacement within line
  - No flags means first occurence

sed – substitute Samples

- sed 's/ists/ISTs/'
  - Replaces first ist
- sed 's/ists/ISTs/g'
  - Replaces global (flag=g) within line
- sed 's/ists/ISTs/2'
  - Replaces second occurence within line
- sed 's/ists//g'
  - Removes all ists from all lines

sed – substitute back references

- Parts of regular expressions may be reused in the new added text
- & adds whole regular expression
- 9 buffers may be used
  - Sub regular expression within \(\)
  - Referenced with \1 - \9
- Example: switch position of 2 tab-separated columns

```
s/\(.*\)\t\(.*\)/\2\t\1/
```

sed – Hold space

- Secondary buffer
  - Transfer between pattern space with commands

- Hold and destroy (h)
  - Overwrites hold space with a copy of pattern space
- Hold and append (H)
  - Appends pattern space to hold space
- Get and destroy (g)
  - Overwrites pattern space with hold space
- Get and append (G)
  - Appends hold space to pattern space
- Exchange
  - Swaps hold space and pattern space

sed –Hold Space Example / 1

- Task: delete text between two words (first,second) that are not in the same line
  - First approach: isolate lines that are spanned by these words
  - Address Range: /BEGIN/,/END/
  - /BEGIN/,/END/d
    - Deletes too much(!), sed works normally line-oriented
- Solution:
  1. Accumulate all lines from /BEGIN/ to /END/ into hold space
  2. Copy/Exchange hold space to pattern space
  3. Substitute within this pattern space (remove /BEGIN.*END/)
- Only /BEGIN/ and /END/ are known!
  - Add line with /BEGIN/
  - Add lines between /BEGIN/ and /END/
  - Add line with /END/

sed –Hold Space Example / 2

- Put line with /BEGIN/ in hold space
  - /BEGIN/{

```
        h        # overwrite hold space
        d        # delete pattern space
    }
```

    - Hold space Contains line with /BEGIN/, pattern space empty
- Append lines without /END/ in hold
  - /END/! {

```
        H        # append each line to hold space
        d        # delete pattern space
    }
```

    - Hold space contains line with /BEGIN/, and lines before /END/

sed –Hold Space Example / 3

- Exchange hold space and pattern space

```
/END/{
  x
   G # append hold (END line) to pattern
}
# pattern space contains now all lines
s/BEGIN.*END//
```

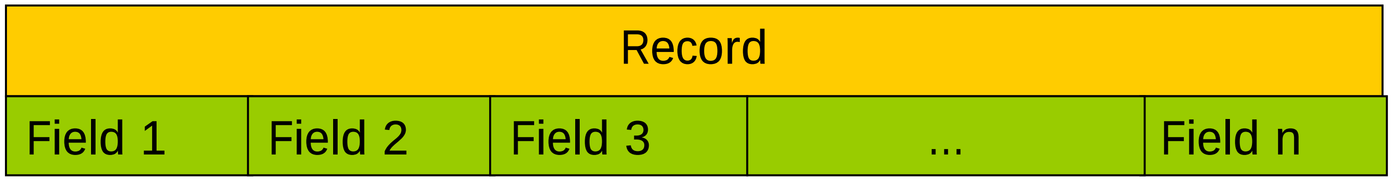
awk

- awk=
  - Aho, Alfred V.
  - Weinberger, Peter J.
  - Kernigham, Brian W.
- Treats files as collection of records and fields

Awk- input file

```
93111111        Meier Mustermann   526
05222222        Susi Malermeister  534
98765432        Hubsi Müller       937
```

Awk – basics



- Iterates over records
- Records are read from file and stored into a record buffer
  - Called $0
- Fields can be referenced by $1, ... $n

Awk – Script Layout

```
BEGIN      { Initial Processing Action}
Pattern1   {Action}
Pattern2   {Action}
Pattern3   {Action}
  ...
END        { End Processing Action }
# each part is optional!
```

Awk – Begin Processing

- Initial processing is done ONCE
  - BEFORE awk starts reading the file
  - Used for setting awk variables
  - Used for printing output headers

Awk – Body Processing

- Data in a file is processed in a loop

```
foreach record do
   foreach action pattern
        if (pattern matches current-record)
                apply-action to record
        end
end
```

Awk - Patterns

- Simple Patterns
  - BEGIN, END
  - "No pattern" means apply always
- Regular Expressions
  - ~ matches text: $0 ~ /regexp/
  - !~ must not match text $2 !~ /otherregexp/
- Arithmetic Expressions (+,-,*,/,...)
  - Matches when expressions evaluates not to 0: $3 + $1 - $4

Awk – Combined Patterns

- Patterns may be combined with
- Relational Expressions
  - ==,!=,<,>,>=,<=
- Logical Expressions
  - !,&&,|| as in Java
- Range Patterns
  - Start-pattern, end-pattern

Awk – end processing

- Invoked once after all input data has been read and all actions have been invoked

Awk - Sample

- Adds numbers in a file

```
BEGIN { print "Gesamtsumme"
                total = 0}
                 { total += $1 }
END { print "------"
            print "Total Sales", total }
```

Awk - Statements

- print
  - Prints Text and variables
  - When separated via , printed into fields of output format
  - Formatted print with printf or sprintf (see C language)
- Variable assignment
  - name=value
- Variable usage
  - With its name
  - Within strings $name
  - Fields with $1 to $n, Records with $0

Awk – Control constructs

- if-else
- next
  - Skips processing of record
    - Like continue in Java
- getline
  - Reads next record but continues processing at current script position
- Loops
  - while,do-while,for
- Associative arrays
  - Similar to Java Hashtable

Record types

- Awk controls the format of records and fields with predefined variables
- FS = Input Field Separator
- RS = Input Record Separator
- OFS = Output Field Separator
- ORS = Output Record Separator
- May be changed in BEGIN block

Other record types

```
BEGIN { FS=","
            OFS=":"
}
      { print $1, $2, $3 }
```

Awk – Predefined variables

- NF – number of non empty fields within a record
- NR – number of records read from all files
- FNR – number of records read in current file
- FILENAME – name of current file

Awk - Functions

- String functions
  - Length, Index, Substring, Split, substitution, global substitution
- Mathematical
  - int – round to integer
  - rand,srand (random numbers),
  - Angle functions (cos,sin,...)
- User defined functions
  - Example:

```
function myOwnFunction(x,y) {
        return x;
}
```

Awk – system functions

- Calling System functions from awk scripts
  - Syntax: system("UNIX command")
  - Return value is exit code

```
BEGIN {
   if (system ("wc input.txt") != 0) {
    ....
   }
}
```

Literatur

- Shells and Unix Commands
  - Forouzan and Gilberg: *UNIX and Shell Programming*, Brooks/Cole
  - Peek et al: *UNIX Power Tools*, O'Reilly
  - Newham and Rosenblatt: *Learning the bash Shell*, O'Reilly
  - GNU Bash Manual: http://www.gnu.org/software/bash/manual/bash.html
- sed & awk
  - Dougherty & Robbins: sed & awk, O'Reilly
  - GNU awk http://www.gnu.org/software/gawk/manual/
  - GNU sed http://www.gnu.org/software/sed/manual/sed.html

Summary / 1

- Shells
  - Link between end user and operating system
  - Power comes from chaining Unix commands
    - File system operations
    - Filters
    - Scripts build new commands
  - Command Line Processing

Summary / 2

- Regular Expressions
  - grep
- Editors
  - Sed
    - Line based
  - Awk
    - Record & Field based