bash

How it works and how to (not) use it

Red Hat
Roman Rakus <rrakus@redhat.com>
May 21, 2011

Part I

How it works?

What is bash

Section 1

Introduction

shell

- Macro processor
- Unix shell – command interpreter and programming language
- Interactive or non-interactive
- Sync, async, redirection
- Builtins

bash

- Shell
- Command language interpreter
- Bourne-Again SHell
- IEEE POSIX
- tcsh, ksh, zsh, dash, . . .

Section 2

Shell syntax

Shell operation

1. Read input
   - file
   - string
   - terminal
2. Break input into words and operators
   - metacharacters – `| \ & ; ( ) < > space tab`
   - quoting
3. Parse tokens to commands
4. Shell expansions
5. Redirections
6. Execute the command
7. Wait for the command

fedora

Quoting

- Escape character – `\`
  - `\\ \" \newline`
- Single Quotes – `' '`
  - `'$variable' '\"' '''`
- Double Quotes – `" "`
  - `' $ \`
  - `"$variable $variable \""`
- ANSI-C Quoting
  - `$'\r' $'\nnn' $'\xHH' $'\uHHHH'`
- Locale-Specific Translation
  - `$"Hello world"`

Comments

- `# whatever...`

Section 3

Shell Commands

Simple Commands

- most often
- return status
  - exit status, *waitpid()*
  - 128+n for signal n

Pipelines

- `|` or `|&`
- executed in subshells
- return status
  - exit status of last command
  - `pipefail` option – rightmost

Lists of Commands

- pipelines separated by `&&`, `||`, `;`, `&`
- optionally terminated by `;`, `&`, or newline
- `&`
  - run command asynchronously in a subshell (background)
  - return status – 0
- `;`
  - run command sequentially
  - return status – exit status of last command
- `&&`, `||`
  - `cmd1 && cmd2` or `cmd1 || cmd2`
  - depends on exit status of cmd1
  - return status – exit status of last command

Compound Commands

- Looping Constructs
- Conditional Constructs
- Grouping Commands
- redirections apply to all within compound

Looping Constructs

`until test-commands; do consequent-commands; done`

- exit status of test-commands is not zero
- return status – last command or 0 if none executed

`while test-commands; do consequent-commands; done`

- exit status of test-commands is zero
- return status – last command or 0 if none executed

Looping Constructs

`for name [ [in [words ...] ] ; ] do commands; done`

- expand words and execute command for each
- expand to each possitional parameters – `"$@"`
- return status – last command or 0 if none executed

`for name (( expr1 ; expr2 ; expr3 )) ; do commands; done`

`break`, `continue`

Conditional Constructs

```
if test-commands; then consequent-commands;
[elif more-test-commands; then more-consequents;]
[else alternate-consequents;]
```

- return status – last command or 0 if none executed

```
case word in [[(] pat [| pat]...) command-list ;;] esac
```

- nocasematch shell option
- clause terminated with `;;`, `;&`, `;;&`
- return status – last command or 0 if no pattern matched

Conditional Constructs

`(( expression ))`

- arithmetic expression
- let "expression"
- return status – value of expression – 0 on non-zero else 1

Conditional Constructs

`[[ expression ]]`

- conditional expression
- Word splitting and filename expansion are not performed.
- `<` and `>` sort lexicographically with current locale
- `==` and `!=` pattern matching
- `=~` extended regular expression *(regex(3))*
  - return value 0 if matched, 1 otherwise, 2 syntax error
  - Quote to force matching as a string.
  - Parethesized subexpressions saved in `BASH_REMATCH` variable.
- `( expression )`, `!`, `&&`, `||`

Grouping Commands

- `( list )` - subshell
- `{ list; }` - current shell

Section 4

Shell Functions

Shell Functions

- `name () compound-command [ redirections ]`
- `unset -f` deletes function definition
- exit status – 0 on successfull definition, last command on execution
- arguments are positional parameters
- `return` builtin, `RETURN` trap
- `typeset -f`, `typeset -F` (or `declare`) bulitin lists all functions

Section 5

Shell Parameters

Shell Parameters

- parameter is entity that stores values
- variable is parameter denoted by a name
- variable has a value and one or more attributes – `declare` builtin

Positional Parameters

- `${N}` or `$N`, `N` is one or more digits
- `$#` – number of possitional parameters
- cannot assign to them
- `set`, `shift` builtins

Special Parameters

- `*` – positional parameters, `"$*"` ⇒ `"$1c$2c..."`, `IFS`
- `@` – positional parameter, `"$@"` ⇒ `"$1" "$2"...`
- `#` – number of positional parameters
- `?` – exit status of the most recently executed foreground pipeline
- `$` – PID of the shell, in () subshell it's invoking shell
- `0` – name of the shell or shell script

Section 6

Shell Expansion

Shell Expansion

- Performed on the command line after it has been split into tokens
- Several types, done in the order
  1. brace expansion – change number of words
  2. tilde expansion
  3. parameter and variable expansion
  4. arithmetic expansion
  5. process substitution
  6. command substitution
  7. word splitting – change number of words
  8. filename expansion – change number of words
- Quote removal is performed after all expansion

Brace Expansion

- similar to filename expansion
- `a{d,c,b}e` ⇒ `ade ace abe`
- sequence expression `{x..y[..incr]}`
  - `x` and `y` are integers or single character
  - `incr` is optional increment, integer
  - intgeres can be prefixed with 0
  - `a{a..d..2}` ⇒ `aa ac`
  - `{10..01..-2}` ⇒ `10 08 06 04 02`

Tilde Expansion

expression	result
`~`	`$HOME`
`~/foo`	`$HOME/foo`
`~fred/foo`	subdir foo of the home dir of the user fred
`~+/foo`	`$PWD/foo`
`~-/foo`	`$OLDPWD/foo`
`~N`	`’dirs +N’`
`~+N`	`’dirs +N’`
`~-N`	`’dirs -N’`

Shell Parameter Expansion

- `$parameter`, `${parameter}`
- `${!parameter}` – indirect expansion
- In all parameter expansions, `:` can be ommited. Without `:` bash not test parameter to null (a="")
- `word` is subject to tilde expansion, parameter expansion, command substitution and arithmetic expansion

fedora

Shell Parameter Expansion

- `${parameter:-word}` – if `parameter` unset (or null) `word` substituted, otherwise `parameter`
- `${parameter:=word}` – if `parameter` unset (or null) `word` assigned to `parameter` and then substituted
- `${parameter:?word}` – if `parameter` unset (or null) expansion of `word` written to stderr, noninteractive shell exits
- `${parameter:+word}` – if `parameter` unset (or null) nothing substituted, otherwise `word`

Shell Parameter Expansion

- `${parameter:offset:length}` – substring, `offset` can be negative (`${a:  -1}`)
- `${!prefix*}` – names of variables starting with `prefix`, can use `@` instead of `*`
- `${!name[*]}` – list of array indices (keys), if not array expands to 0 (or null if unset). Can use `@`

fedora

Shell Parameter Expansion

- `${#parameter}`
  - length of expanded value of `parameter`
  - if `parameter` is `*` or `@` expands to number of positional parameters
  - if `parameter` is array with subscript `*` or `@` expands to number of elements

Shell Parameter Expansion

- `${parameter#word}`, `${parameter##word}`, `${parameter%word}`, `${parameter%%word}`
  - `word` is pattern to remove from `parameter`
  - `#`, `##` remove from beginning
  - `%`, `%%` remove from end
  - `$0`, `${0##*/}`, `${0%/*}`

Shell Parameter Expansion

- `${parameter/pattern/string}`
  - replace longest match of `pattern` with `string` on `parameter`
  - `pattern` begins with
  - / – replace all matches
  - # – must match beginning of `parameter`
  - % – must match end of `parameter`
  - `${0/%bash/ksh}`

Shell Parameter Expansion

- `${parameter^pattern}`, `${parameter^^pattern}`, `${parameter,pattern}`, `${parameter,,pattern}`
  - case modification
  - `^`, `^^` – to upper on first char, every chars
  - `,`, `,,` – to lower on first char, every chars
  - `${0^^}`, `${a[*]^[aeiou]}`

Command Substitution

- `$(command)`, `` `command` ``
- expansion by executing command and replacing standard output
- removes trainling newlines
- `$(cat file)` is equivalent to `$(<file)`
- nesting is easier in `$( )` form

Arithmetic Expansion

- `$((  expression  ))`
- arithmetic rules same as in C language
- `++`, `--`, `+`, `-`, `*`, `/`, `%`, ...
- `a=07; z=$(( a++ ))`

Process Substitution

- Use FIFO or /dev/fd method
- `<(list)`, `>(list)`
- input (`>()`) or output (`<()`) of process connected to FIFO (or /dev/fd)
- expands to file name
- `diff <(command1) <(command2)`
- `tar cf >(bzip2 -c > file.tar.bz2) $directory_name`

Word Splitting

- on results of parameter expansion, command substitution and arithmetic expansion without double quotes
- delimiters are each characters of `$IFS` (`space tab newline`)
- no expansion $\longrightarrow$ no splitting

fedora

Filename Expansion

- can be turned off – `set -f`
- after Word Splitting
- scan each word for `*`, `?` and `[`
- such word is regarded as a pattern. If matched, replaced with alphabetically sorted list, else word is unchanged.
- several options – `nocaseglob nullglob failglob dotglob`

Filename Expansion

Usefull guide to quoting [Quoting]

Quoting example

```
XYZ='abc f*'
grep $XYZ bar
# grep abc foo.1 foo.2 bar
grep "$XYZ" bar
# grep 'abc f*' bar
```

Pattern Matching

- `*` – matches any string
- `?` – matches any single character
- `[...]` – matches collation – `LC_COLLATE`
- ending `/` matches only directories
- option `extglob`

Section 7

Redirections

Redirections

- processed in the order, from left to right
  - `ls > dirlist 2>&1`
  - `ls 2>&1 > dirlist`
- in the following `n` is file descriptor, on `word` are performed expansions
- Redirecting Input – `[n]<word`
- Redirecting Output – `[n]>[|]word`
- Appending Redirected Output – `[n]>>word`
- Redirecting Standard Output and Error – `&>word`
- Appending Standard Output and Error – `&>>word`

Redirections

- Here Documents

```
<<[-]word
    here-document
delimiter
```

Here-document example

```
a=12
cat␣<<-\EOF
123
acbc
␣␣␣␣␣␣␣␣$a
EOF
```

Redirections

- Here Strings – `<<< word`

Here-document example

```
a=12
cat <<< $a
```

- Other redirections – Duplicating File Descriptor, Moving File Descriptor, Opening File Descriptor for Reading and Writing

Section 8

Executing Commands

Executing Commands

- After all expansions
- Variable assignments are not commands
- First word is commands name
- Remaining words are arguments

Command Search and Execution

1. If command doesn't contain slashes, bash tries a function by that name
2. If it's not a function, bash tries builtin
3. If it's not a function nor a builtin and contains no slashes:
   - bash uses hash tables to remember full path name.
   - bash searches in `$PATH` only if the command is not in the table.
4. Search is successful, or command contains slahes, it's executed in a separate execution environment.
5. If the file is not in executable format, and file is not directory, it's assumed to be a shell script.
6. If not run async, bash waits.

Section 9

Shell Builtin Commands

Shell Builtin Commands

```
:  .  break cd continue eval exec exit export getopts
hash pwd readonly return shift test [ times trap
umask unset
alias bind builtin caller command declare echo enable
help let local logout mapfile printf read readarray
source type typeset ulimit unalias
```

Will talk only about some of them.

Builtins

- `. filename [arguments]`
  - Read and execute commands from the `filename` in the current shell context.
  - Equivalent to `source` builtin

Builtins

`eval [arguments]`

- Reread arguments and execute it – do second parsing.
- Useful when need to do another round of parameter substitutions.

Builtins

```
exec [-cl] [-a name] [command [arguments]]
```

- If `command` is suplied, it replaces current shell without creating a new process
- If no `command` is specified, redirections may be used to affect the current shell.

Builtins

`hash [-r] [-p filename] [-dt] [name]`

- Some basic manipulation with hash table of commands
- `hash -r` – forget all locations
- `hash -t name name ...` – list hashed names
- `hash -d name name ...` – forget hashed names

Builtins

`trap [-lp] [arg] [sigspec ...]`

- arg commands are read and executed when shell receives signal sigspec
- sigspec is 0 or EXIT – do arg when shell exits
- sigspec is DEBUG – do arg before every command
- sigspec is RETURN – do arg when function or source builtin finishes
- sigspec is ERR – do arg when command has non-zero exit status.

Builtins

`declare [-aAfFilrtux] [-p] [name[=value] ...]`

- Declare variables and give them attributes.
- `-a` – indexed array
- `-A` – associative array
- `-i` – integer
- `-r` – readonly
- `-t` – trace attribute. Traced functions inherit `DEBUG` and `RETURN` traps
- Functions variables are `local`. Can use `-g` to set them global.

Builtins

`echo [-neE] [arg ...]`

- Output the args separated by spaces, terminated with a newline.
- `-n` – no newline
- `-e` – do interpretation of backslash characters
- `-E` – do not do interpretation of backslash characters

Builtins

`printf [-v var] format [arguments]`

- Write the formatted `arguments` to stdout
- `-v` – assign output to the *var*
- accept same format as `printf(1)` and few more
- %b – expand backslashes
- `%(datefmt)T` – output date-time string, `strftime(3)`
- `printf` is preferred to `echo`

Builtins

```
read [-ers] [-a aname] [-d delim] [-i text]
[-n nchars] [-N nchars] [-p prompt] [-t timeout]
[-u fd] [name ...]
```

- Read one line from stdin, first word is assigned to the first `name`, second word to second `name`, `....` `IFS` used to separate words.
- `-a aname` – assign to array `aname`.
- `-u fd` – read from file descriptor.

Builtins

`type [-afptP] [name ...]`

- Indicate how each `name` would be interpreted.
- `which` is not right way!

Builtins

`set [--abefhkmnptuvxBCEHPT] [-o option-name] [argument ...]`

- Very complicated.
- Allows to change shell options.
- Display the names and values of shell variables.
- Set positional parameters.
- `-n` – Check a script (read but don't execute).
- `-v` – Print input lines as they are read.
- `-x` – Print a trace of a command after expansions and before execution
- - turns on an option, + turns it off

Builtins

`shopt [-pqsu] [-o] [optname ...]`

- Change additional shell optional behavior.
- `-s -u` – set/unset (enable/disable) each `optname`.
- Few options:
    - `checkhash` – Check hashed commands.
    - `globstar` – `**` check also subdirectories in globbing.
    - `nullglob` – Glob which match no file expands to null string.

Section 10

Shell Variables

Shell Variables

- `IFS` – List of characters that separate fields.
- `PATH` – Colon-separated list of dirs for command lookup.
- `PS1` – Primary prompt string.
- `BASHPID` – PID of current Bash process. Better than `$$`
- `LC_{ALL,COLLATE,CTYPE,MESSAGES,NUMERIC}` – Locale specification.
- `PWD` – Current working directory.
- `RANDOM` – Generates random number 0 — 32767

Section 11
Arrays

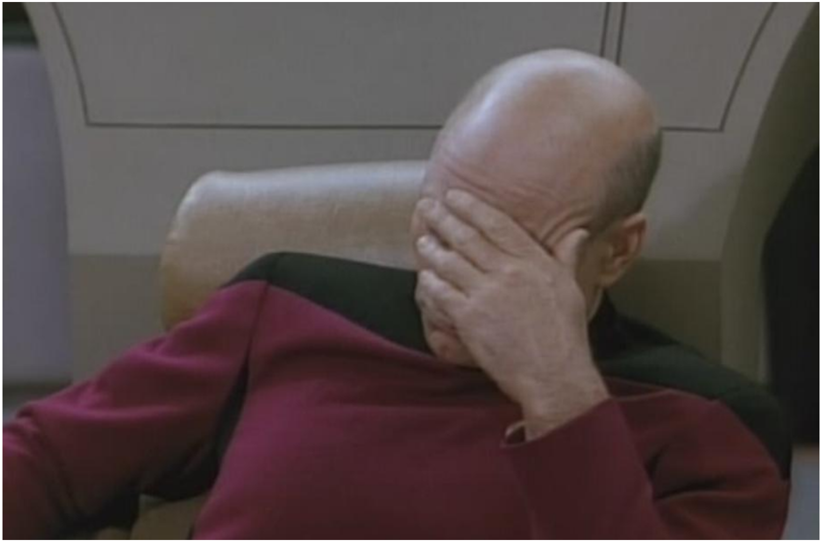
Arrays

Indexed arrays and Associative arrays

- `declare -a name` or `name[subscript]=value`
- `name=(value`*1*` ... value`*n*`)`
- `declare -A name`
- Referencing – `$name[subscript]`
- Subscript `*` and `@` expands to all members.
- In double quotes `*` expands to one word, `@` to *n* words.

Part II

How to (not) use bash



Section 12

echo

echo, the right way?

examples/01-echo/test1.bash

```
#!/bin/bash

var0='-en'
var1='\n'
var2='\'
var3='0123'

echo var0: $var0                #-en
echo var1: $var1                #\n
echo var2: $var2                #\
echo var3: $var3                #0123

echo $var0$var1$var2$var3       #-en\n\0123
echo $var0 $var1$var2$var3      #-en␣\n\0123
echo -e $var0$var1$var2$var3    #-en[newline]\0123
echo -e $var0 $var1$var2$var3   #-en␣[newline]\0123
echo -en $var0$var1$var2$var3   #-en[newline]\0123[nonewline]
echo DONE
```

echo, the right way?

- See `examples/01-echo/test2.bash`.
- Problem is with variables containing '-' and escaped sequences.
- echo is not considered as portable.
- Even POSIX suggests to use printf.

Section 13
globbing

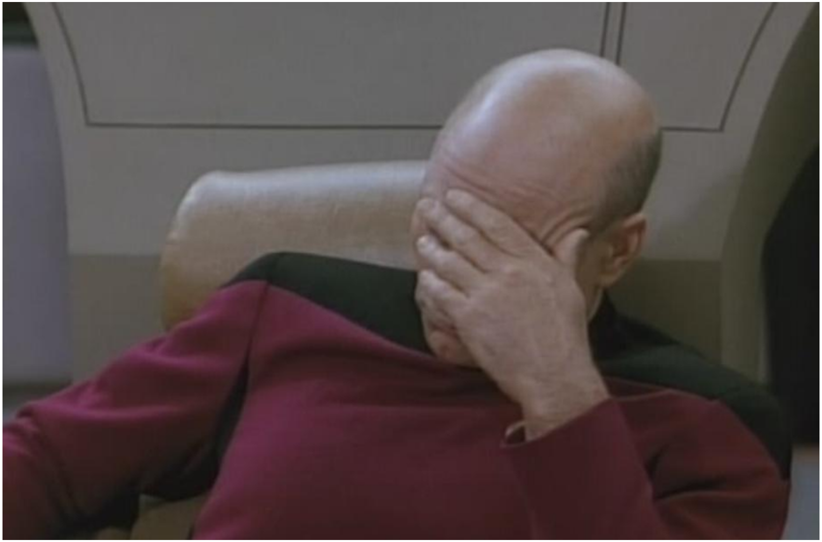
globbing

examples/02–globbing/test1.bash

```
#!/bin/bash

printf 'Listing all files\n'
for file in $(ls *); do
  printf "%s\n" "$file"
done

printf 'Listing all files with *this* in name\n'
for file in $(ls * | grep this); do
  printf "%s\n" "$file"
done
```



globbing

- See `examples/02-globbing/test2.bash`.
- bash is doing globbing, not ls.
- Remember; globbing is done after all expansions and word splitting according to `IFS`.
- `nullglob` and `globstar` are usefull.
- See `examples/02-globbing/test3.bash`.

Section 14
execution

execution

- bash builtins `type` and `hash`, `PATH` variable and command `which`
- See `examples/03-execution/test.bash`

Section 15
redirections

redirections

- Order of redirections is significant. See `examples/04-redirection/test{1,2,3}.bash`
- When is redirection done. See `examples/04-redirection/test4.bash`
- `read` builtin and redirections. See `examples/04-redirection/test5.bash`
- Redirections and file descriptors. See `examples/04-redirection/test6.bash`

Section 16

Parameter expansions

Parameter expansions

- Faster (and pretier) than external commands. See `examples/05-expansion/test1.bash`
- Some good uses of `case`. See `examples/05-expansion/test2.bash`

Section 17

Questions?

Questions?

Bibliography

- Chet Ramey, Brian Fox:
  Bash Reference Manual
  2010
  http://www.gnu.org/software/bash/manual/
- Uwe Waldmann:
  A Guide to Unix Shell Quoting
  2009
  http://www.mpi-inf.mpg.de/~uwe/lehre/unixffb/quoting-guide.html

The end.

Thanks for listening.