Linux Architecture

Shell, Command Interpreter Programming

Shell

- Interpreter | Shell | Cli interface | Programming Language
- command line interface
- what is executed?
- PATH
- full path ; relative path
- alias
- builtin ; keyword - https://askubuntu.com/questions/445749/whats-the-difference-between-shell-builtin-and-shell-keyword
- type

Variables and Types

- **VARIABLE=value**
- **echo $VARIABLE**

https://linux.die.net/Bash-Beginners-Guide/sect_10_01.html

- **declare OPTION(s) VARIABLE=value**

Option	Meaning
-a	Variable is an array.
-f	Use function names only.
-i	The variable is to be treated as an integer; arithmetic evaluation is performed when the variable is assigned a value.
-p	Display the attributes and values of each variable. When -p is used, additional options are ignored.
-r	Make variables read-only. These variables cannot then be assigned values by subsequent assignment statements, nor can they be unset.
-t	Give each variable the *trace* attribute.
-x	Mark each variable for export to subsequent commands via the environment.

- **readonly TUX=*penguinpower***
- local

Variables and Substitution

- **${parameter}**
- **${parameter-default}, ${parameter:-default}**
- **${parameter=default}, ${parameter:=default}**
- **${parameter+alt_value}, ${parameter:+alt_value}**
- **${parameter?err_msg}, ${parameter:?err_msg}**
- **${#var}**
- **${var#Pattern}, ${var##Pattern}**
- **${var%Pattern}, ${var%%Pattern}**
- **${var:pos:len}**
- **${var/Pattern/Replacement}**
- **${var//Pattern/Replacement}**

https://tldp.org/LDP/abs/html/parameter-substitution.html

Loops

```
for VARIABLE in file1 file2 file3
do
   command1 on $VARIABLE
   command2
   command
done
```

```
while command
do
   echo a
   a=a+1
   echo ok
done
```

```
until command
do
   echo a
   a=a+1
   echo ok
done
```

Functions

```
function_name ()
{
   commands
}
```

```
function function_name { commands; }
```

Conditional Statements

```
command1 && truecommand || falsecommand
```

```
if [ conditional expression1 ]
then
    statement1
    statement2
elif [ conditional expression2 ]
then
    statement3
    statement4
else
    statement5
fi
```

Conditional Statements

```
case "$1" in
      start)
          start
          ;;
      stop)
          stop
          ;;
      status)
          status anacron
          ;;
      restart)
          stop
          start
          ;;
      condrestart)
          if test "x`pidof anacron`" != x; then
              stop
              start
          fi
          ;;
      *)
          echo $"Usage: $0 {start|stop|restart|condrestart|status}”
          exit 1
esac
```

Arguments

- Available in functions and in scripts.
- $1, $2, $n
- $*, $@
- $#

```
i=1;
j=$#;
while [ $i -le $j ]
do
   echo "Username - $i: $1";
   i=$((i + 1));
   shift 1;
done
```

Options

```
while getopts ":ht" opt; do
   case ${opt} in
      h ) # process option h
        ;;
      t ) # process option t
        ;;
      \? ) echo "Usage: cmd [-h] [-t]”
        ;;
    esac
done
```

trap

```
tempfile=/tmp/tmpdata
trap "rm -f $tempfile" EXIT

function cleanup()
{ # ... }
trap cleanup EXIT
```

https://www.linuxjournal.com/content/bash-trap-command

https://opensource.com/article/20/6/bash-trap

read

- https://www.computerhope.com/unix/bash/read.htm

exec

- exec 3<>/dev/tcp/www.google.com/80
  echo -e "GET / HTTP/1.1\r\nhost: http://www.google.com\r\nConnection: close\r\n\r\n" >&3
  cat <&3

- cat </dev/tcp/time.nist.gov/13

select

```
calculate () {
  read -p "Enter the first number: " n1
  read -p "Enter the second number: " n2
  echo "$n1 $1 $n2 = " $(bc -l <<< "$n1$1$n2")
}

PS3="Select the operation: ”

select opt in add subtract multiply divide quit;
do
   case $opt in
      add)
         calculate "+";;
      subtract)
         calculate "-";;
      multiply)
         calculate "*";;
      divide)
         calculate "/";;
      quit)
         break;;
      *)
         echo "Invalid option $REPLY";;
   esac
done
```

shell bash

- [[is not available in sh (only [which is more clunky and limited). See also Difference between single and double square brackets in Bash
- sh does not have arrays.
- Some Bash keywords like local, source, function, shopt, let, declare, and select are not portable to sh. (Some shimplementations support e.g. local.)
- Bash has many C-style syntax extensions like the three-argument for((i=0;i<=3;i++)) loop, += increment assignment, etc. The $'string\nwith\tC\aescapes' feature is tentatively accepted for POSIX (meaning it works in Bash now, but will not yet be supported by sh on systems which only adhere to the current POSIX specification, and likely will not for some time to come).
- Bash supports <<<'here strings'.
- Bash has *.{png,jpg} and {0..12} brace expansion.
- ~ refers to $HOME only in Bash (and more generally ~username to the home directory of username).This is in POSIX, but may be missing from some pre-POSIX /bin/sh implementations.
- Bash has process substitution with <(cmd) and >(cmd).
- Bash has Csh-style convenience redirection aliases like &| for 2>&1 | and &> for > ... 2>&1
- Bash supports coprocesses with <> redirection.
- Bash features a rich set of expanded non-standard parameter expansions such as ${substring:1:2}, ${variable/pattern/replacement}, case conversion, etc.
- Bash has significantly extended facilities for shell arithmetic (though still no floating-point support). There is an obsolescent legacy $[expression] syntax which however should be replaced with POSIX arithmetic $((expression)) syntax. (Some legacy pre-POSIX sh implementations may not support that, though.)
- Magic variables like $RANDOM, $SECONDS, $PIPESTATUS[@] and $FUNCNAME are Bash extensions.
- Syntactic differences like export variable=value and ["x" == "y"] which are not portable (export variable should be separate from variable assignment, and portable string comparison in [...] uses a single equals sign).
- Many, many Bash-only extensions to enable or disable optional behaviour and expose internal state of the shell.
- Many, many convenience features for interactive use which however do not affect script behaviour.

/bin/bash options

https://tldp.org/LDP/abs/html/options.html

Abbreviation	Name	Effect
-B	brace expansion	*Enable* brace expansion (default setting = *on*)
+B	brace expansion	*Disable* brace expansion
-C	noclobber	Prevent overwriting of files by redirection (may be overridden by >|)
-D	(none)	List double-quoted strings prefixed by $, but do not execute commands in script
-a	allexport	Export all defined variables
-b	notify	Notify when jobs running in background terminate (not of much use in a script)
-c ...	(none)	Read commands from **...**
checkjobs		Informs user of any open jobs upon shell exit. Introduced in version 4 of Bash, and still "experimental." *Usage:* shopt -s checkjobs (*Caution:* may hang!)
-e	errexit	Abort script at first error, when a command exits with non-zero status (except in until or while loops, if-tests, list constructs)
-f	noglob	Filename expansion (globbing) disabled
globstar	*globbing* star-match	Enables the ** globbing operator (version 4+ of Bash). *Usage:* shopt -s globstar
-i	interactive	Script runs in *interactive* mode
-n	noexec	Read commands in script, but do not execute them (syntax check)
-o Option-Name	(none)	Invoke the *Option-Name* option
-o posix	POSIX	Change the behavior of Bash, or invoked script, to conform to POSIX standard.
-o pipefail	pipe failure	Causes a pipeline to return the exit status of the last command in the pipe that returned a non-zero return value.
-p	privileged	Script runs as "suid" (caution!)
-r	restricted	Script runs in *restricted* mode (see Chapter 22).
-s	stdin	Read commands from stdin
-t	(none)	Exit after first command
-u	nounset	Attempt to use undefined variable outputs error message, and forces an exit
-v	verbose	Print each command to stdout before executing it
-x	xtrace	Similar to -v, but expands commands
-	(none)	End of options flag. All other arguments are positional parameters.
--	(none)	Unset positional parameters. If arguments given (-- *arg1 arg2*), positional parameters set to arguments.

Basic Regular Expression

- . (dot) - match one character
- * (asterisk) - match zero or more occurrences of the preceding pattern
- .* - match any number of any characters

```
$ grep "root" /etc/passwd
root:x:0:0:root:/root:/bin/bash
$ grep "r.t" /etc/passwd
$ grep "r.*t" /etc/passwd
root:x:0:0:root:/root:/bin/bash
list:x:38:38:Mailing List Manager:/var/list:/usr/sbin/nologin
gnats:x:41:41:Gnats Bug-Reporting System (admin):/var/lib/gnats:/usr/sbi
systemd-timesync:x:100:102:systemd Time Synchronization,,,:/run/systemd:
systemd-network:x:101:103:systemd Network Management,,,:/run/systemd/net
systemd-resolve:x:102:104:systemd Resolver,,,:/run/systemd/resolve:/bin/
systemd-bus-proxy:x:103:105:systemd Bus Proxy,,,:/run/systemd:/bin/false
pollinate:x:111:1::/var/cache/pollinate:/bin/false
postgres:x:998:1003::/home/postgres:
```

Basic Regular Expression

- **^** (caret) - match text at the beginning of a line
- **$** (dollar sign) - match text at the end of a file

```
$ grep "^r" /etc/passwd
root:x:0:0:root:/root:/bin/bash
$ grep "bash$" /etc/passwd
root:x:0:0:root:/root:/bin/bash
bitnami:x:1000:1000:Ubuntu:/home/bitnami:/bin/bash
ubuntu:x:1000:1000::/home/bitnami:/bin/bash
```

- **[ ]** (square brackets) - specifies a range, match one of the characters in brackets
  If you did *m[aou]m* it could become: *mam, mum, mom*
  if you did: *m[a-d]m* it can become anything that starts and ends with m and has any character a to d in between.
- **[^]** - This construct is similar to the [ ] construct, except rather than matching any characters inside the brackets, it'll match any character, as long as it is not listed between the [and]. This is a logical NOT.

```
$ grep "[uU]buntu" /etc/passwd
bitnami:x:1000:1000:Ubuntu:/home/bitnami:/bin/bash
ubuntu:x:1000:1000::/home/bitnami:/bin/bash
$ grep "systemd[^ ]" /etc/passwd
systemd-timesync:x:100:102:systemd Time Synchronization,,,:/run/systemd:
systemd-network:x:101:103:systemd Network Management,,,:/run/systemd/net
systemd-resolve:x:102:104:systemd Resolver,,,:/run/systemd/resolve:/bin/
```

Basic Regular Expression

- **\(\)** (round brackets) - group a part of the regular expression together, could be used with curly brackets
- **\{x, y\ }** (curly brackets) - match at least x occurrences, but not more than y occurrences of the preceding pattern. Also possible: **\{x\}** – match x occurrences, **\{x,\}** – match at least x occurrences, **\{,x\}** – match not more the x occurrences
- **** (backslash) is used as an "escape" character, i.e. to protect a subsequent special character. Thus, "\\" searches for a backslash. Note you may need to use quotation marks and backslash(es).

```
$ echo "abracadabra" | grep "a[abcd]"
abracadabra
$ echo "abracadabra" | grep "a[abcd]\{2\}"
abracadabra
$ echo "abracadabra" | grep "a[abcd]\{3,\}"
abracadabra
$ echo "abracadabra" | grep "\(a[a-d]\)\{2\}"
abracadabra
$ echo "abracadabra" | grep "\(a[a-d]\)\{,3\}"
abracadabra
```

Links

- https://www.youtube.com/watch?v=x2U9TsqSKmw – Youtube D. Ketov Polytechnical University
  https://www.youtube.com/watch?v=WVHC5Ggl7k4 – Youtube D. Ketov Polytechnical University

- https://github.com/orasul/bash-scripts - some pure bash scripts
- https://github.com/dylanaraps/pure-bash-bible - lot's of pure bash scripts

THE END