Optimized Bash

Use New Bash 4 Functions and Builtins To Write Faster Scripts

TÜBIX 2016, 11 June, Gerik Huland

Use pure bash!

Speed up scripts

Less CPU time by reducing forking and sub processing. Avoid calling helper programs like sed, awk, grep, tr as they can be expensive.

Speed up scripts

Use the power of bash

Don't bother learning awk and sed. Miss out to learn dozens of options to tr, xargs, grep etc. We often see code snippets like this

```
du -bc . |grep total|awk '{print $1}'
```

This is because people use awk only for line splitting. Using it properly would be

```
du -bc . |awk '$2=="total"{print $1}'
```

Pure Bash: Use array for splitting *

```
while read -a out; do
  [ "${out[1]}" = total ] && break
done < <(du -bc .); print ${out[0]}
```

Yes this more lengthy. Otherwise learn proper awk! :)

* will be explained in more detail later

How to start?

Check out new builtins and abilities of Bash 4

Just an assortment

Regular Expressions

```
# Best practice: put regex in a variable
$ regex="(^[^:]+): "
$ [[ message: this is Bash 4.0 =~ $regex ]]
$ echo ${BASH_REMATCH[1]}
message
```

Just an assortment

Regular Expressions

```
# Best practice: put regex in a variable
$ regex="(^[^:]+): "
$ [[ message: this is Bash 4.0 =~ $regex ]]
$ echo ${BASH_REMATCH[1]}
message
```

Associative Arrays

```
$ ARRAY=( [this]="that" [here]="there" )
$ echo ${ARRAY[this]}
that
```

Just an assortment

Regular Expressions

```
# Best practice: put regex in a variable
$ regex="(^[^:]+): "
$ [[ message: this is Bash 4.0 =~ $regex ]]
$ echo ${BASH_REMATCH[1]}
message
```

Associative Arrays

```
$ ARRAY=( [this]="that" [here]="there" )
$ echo ${ARRAY[this]}
that
```

Upper/Lower Case

```
$ a="lower UPPER"; echo ${a^}
Lower UPPER
$ echo ${a^^}
LOWER UPPER
$ echo ${a,,}
lower upper
```

Here Strings

```
$ $a="some text"
$ read b <<<"$a"
$ echo $b
some
```

More

Here Strings

```
$ $a="some text"
$ read b <<<"$a"
$ echo $b
some
```

Less typing for redirects

```
# one expression for stdout and stderr
$ some_program |& grep "some stuff on stderr"
$ rmdir a_dir >& /dev/null
```

And Now Really Speed Up Scripts!

Avoid forking

Keep Clear of Pipes

This doesn't work!

```
cat somefile|while read a; do
    [ "$a" = "some text" ] && break
done
echo $a
```

nothing gets printed! $a's scope is in the subshell only.

This works!

```
while read a; do
    [ "$a" = "some text" ] && break
done <somefile
echo $a
> some text
```

Even with command output

```
...
done < <(some_command)
```

Avoid Forking

Keep Clear of Pipes

Use IFS For Splitting

No need for awk's line splitting!

```
# Save IFS
OLDIFS="$IFS"
# Split on colon, read into array
IFS=":"
while read -a line; do
    echo ${line[0]}
done < some_file
IFS="$OLDIFS"
```

Avoid Forking

Keep Clear of Pipes

Use IFS For Splitting

Eval is cheaper than backticks. Really?

Really! Backticks (or $()) fork a subprocess!

- No Problem if you run an external program (forks anyway)
- Consider when you use bash builtins or functions, e.g.
- Take a look here http://www.fvue.nl/wiki/Bash:_Passing_variables_by_reference
- Consider using upvar (check it out in the bash-completion code)

```
f() { local b; g b; echo $b; }                  # (1)
g() { local "$1" && upvar $1 bar; }             # (2)
f  # Ok: b=bar
```

- In upvar eval is used for variable assignment in a secure way.

But I Just Wanted To Write This Little One Liner!

Just do it.

Dont bother optimizing shell code when writing one liners.

- Take into account the platform your code runs on
  - low power/embedded will be more thankful
  - but their might be only busybox and a few gnu tools
- Looping over 100,000 lines of input will show more benefit
- Happily use new functions and builtins that make your code faster to type
  - `b=${a^^}`
    instead of
    `b=$(echo $a|tr '[[:lower:]]' '[[:upper:]]')`
- If you need associative arrays. My personal advice:
  - consider using more elaborated scripting languages like
    - perl
    - python
    - ruby
    - etc.

And Now: Happy Bashing!

Slideshow created using remark.