


Words...

 **Fix Me!** This article needs a review, it covers two topics (command line splitting and word splitting) and mixes both a bit too much. But in general, it's still usable to help understand this behaviour, it's "wrong but not wrong".

One fundamental principle of Bash is to recognize words entered at the command prompt, or under other circumstances like variable-expansion.

Splitting the commandline

Bash scans the command line and splits it into words, usually to put the parameters you enter for a command into the right C-memory (the `argv` vector) to later correctly call the command. These words are recognized by splitting the command line at the special character position, **Space** or **Tab** (the manual defines them as **blanks**). For example, take the `echo` program. It displays all its parameters separated by a space. When you enter an `echo` command at the Bash prompt, Bash will look for those special characters, and use them to separate the parameters.

You don't know what I'm talking about? I'm talking about this:

```
$ echo Hello little world
Hello little world
```

In other words, something you do (and Bash does) everyday. The characters where Bash splits the command line (SPACE, TAB i.e. blanks) are recognized as delimiters. There is no null argument generated when you have 2 or more blanks in the command line. **A sequence of more blank characters is treated as a single blank.** Here's an example:

```
$ echo Hello           little           world
Hello little world
```

Bash splits the command line at the blanks into words, then it calls `echo` with **each word as an argument**. In this example, `echo` is called with three arguments: "Hello ", "little " and "world "!

Does that mean we can't echo more than one Space? Of course not! Bash treats blanks as special characters, but there are two ways to tell Bash not to treat them special: **Escaping** and **quoting**.

Escaping a character means, to **take away its special meaning**. Bash will use an escaped character as text, even if it's a special one. Escaping is done by preceeding the character with a backslash:

```
$ echo Hello\ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ little \ \ \ \ \ \ \ \ \ \
\ \ \ \ \ \ \ \ \ world
Hello                little                world
```

None of the escaped spaces will be used to perform word splitting. Thus, echo is called with one argument: "Hello little world".

Bash has a mechanism to "escape" an entire string: **Quoting**. In the context of command-splitting, which this section is about, it doesn't matter which kind of quoting you use: weak quoting or strong quoting, both cause Bash to not treat spaces as special characters:

```
$ echo "Hello                little                world"
Hello                little                world
```

```
$ echo 'Hello                little                world'
Hello                little                world
```

What is it all about now? Well, for example imagine a program that expects a filename as an argument, like cat. Filenames can have spaces in them:

```
$ ls -l
total 4
-rw-r--r-- 1 bonsai bonsai 5 Apr 18 18:16 test file

$ cat test file
cat: test: No such file or directory
cat: file: No such file or directory

$ cat test\ file
m00!

$ cat "test file"
m00!
```

If you enter that on the command line with Tab completion, that will take care of the spaces. But Bash also does another type of splitting.

Word splitting

For a more technical description, please read the article about word splitting!

The first kind of splitting is done to parse the command line into separate tokens. This is what was described above, it's a pure **command line parsing**.

After the command line has been split into words, Bash will perform expansion, if needed - variables that occur in the command line need to be expanded (substituted by their value), for example. This is where the second type of word splitting comes in - several expansions undergo **word splitting** (but others do not).

Imagine you have a filename stored in a variable:

```
MYFILE="test file"
```

When this variable is used, its occurrence will be replaced by its content.

```
$ cat $MYFILE
cat: test: No such file or directory
cat: file: No such file or directory
```

Though this is another step where spaces make things difficult, **quoting** is used to work around the difficulty. Quotes also affect word splitting:

```
$ cat "$MYFILE"
m00!
```

Example

Let's follow an unquoted command through these steps, assuming that the variable is set:

```
MYFILE="THE FILE.TXT"
```

and the first review is:

```
echo The file is named $MYFILE
```

The parser will scan for blanks and mark the relevant words ("splitting the command line"):

Initial command line splitting:

Word 1	Word 2	Word 3	Word 4	Word 5	Word 6
echo	The	file	is	named	\$MYFILE

A parameter/variable expansion is part of that command line, Bash will perform the substitution, and the word splitting on the results:

Word splitting after substitution:

Word 1	Word 2	Word 3	Word 4	Word 5	Word 6	Word 7
echo	The	file	is	named	THE	FILE.TXT

Now let's imagine we quoted \$MYFILE , the command line now looks like:

```
echo The file is named "$MYFILE"
```

Word splitting after substitution (quoted!):

Word 1	Word 2	Word 3	Word 4	Word 5	Word 6
echo	The	file	is	named	THE FILE.TXT

Bold Text*72i* love this world

See also

- Internal: Quoting and character escaping
- Internal: Word splitting
- Internal: Introduction to expansions and substitutions
- External: Grymore: Shellquoting (<http://www.grymoire.com/Unix/Quote.html>)

Discussion

kobu, [2012/03/27 21:25 \(\)](#)



space, tab AND newline ... don't forget newline

Rptx, [2014/06/05 03:39 \(\)](#)

Doesn't parameter expansion happen before word splitting? The last part is wrong.
\$MYFILE will be expanded before word splitting.

Jan Schampera, [2015/06/18 04:25 \(\)](#)

But the quotes are there to preserve the word boundaries (which describes exactly the job of quoting)

 [syntax/words.txt](#)  Last modified: 2020/12/11 16:14 by karan_sharma

This site is supported by Performing Databases - your experts for database administration

Bash Hackers Wiki



Except where otherwise noted, content on this wiki is licensed under the following license:
GNU Free Documentation License 1.3

