

Power Shell Usage

Simon Myers
simon.myers@gbdirect.co.uk
GBdirect Ltd

UKUUG Linux 2003 Conference • August 2003
<http://www.ukuug.org/events/linux2003/>

1 Intro

- Tips for driving Bash better in everyday Linux use
- Target audience: people who type things in Linux



- Assumptions:
 - You suspect Bash has features useful to you
 - You know that *bash(1)* is *loong*
 - You're too lazy to read it

2 Not a List of Keystrokes

- More interesting tips than just listing keystrokes
- This talk not about:
 - Ctrl+W, Meta+BkSpc, Meta+D, Ctrl+K, Ctrl+U, Meta+F, Meta+B, Shift+Ins, Ctrl+T, Ctrl+], Ctrl+Meta+]



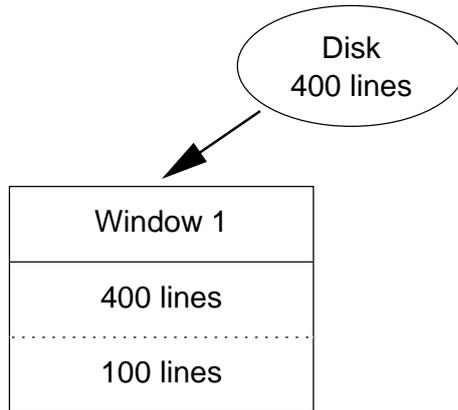
- If you want to learn keystrokes, look them up

3 History

- Command history means never having to retype previous commands
- In theory
- Default configuration makes this not always the practice
- Problems:
 - Sometimes commands don't seem to get saved
 - It can be awkward to find those that have been

4 The 'New Window' Problem

- Opening a terminal might read 400 history lines off disk

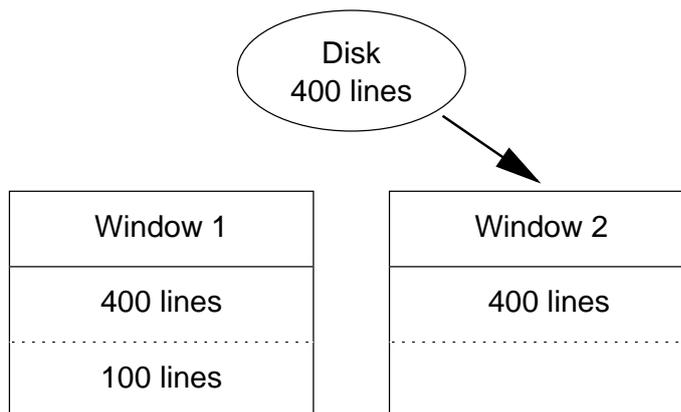


- Type 100 new command lines into it
- One command is taking a while to finish...

For example a long compilation, or an FTP or SSH session, and you need to do something else local too.

5 The 'New Window' Problem

- So open a second window

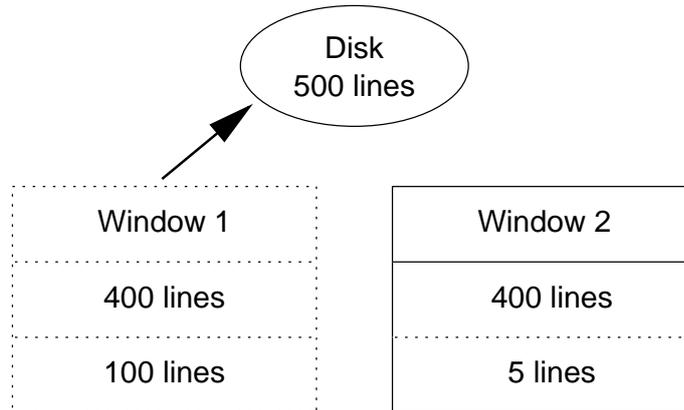


- The new window doesn't have the 100 most recent lines!
- History is saved on exit — and the first shell hasn't exited yet

The history on disk still has just the original 400 lines, so that's what gets loaded into the 2nd window.

6 'Loser Takes All'

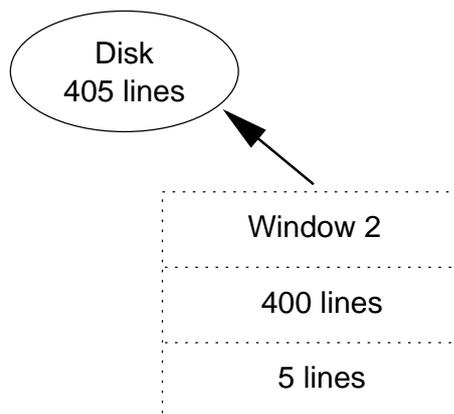
- Now closing the first window will write the history to disk



- All 500 lines saved for future reference
- But that second window, with only 5 lines typed in it, is still open...

7 'Loser Takes All'

- Closing the second window saves *its* history to disk



- The 100 lines from the first window are overwritten!
- History is only remembered from the shell that finishes last

8 Rewriting History

- Both of these problems stem from only writing history lines to disk when a shell exits
- The solution is to append each line to the history individually
- Specify this in *.bashrc*
 - Make Bash append rather than overwrite the history on disk:

```
shopt -s histappend
```
 - Whenever displaying the prompt, write the previous line to disk:

```
PROMPT_COMMAND='history -a'
```
- A new shell gets the history lines from all previous shells

If you've configured your terminal windows to run log-in shells then you might need to use *.bash_profile* instead of *.bashrc*. But on many systems *.bash_profile* sources *.bashrc* anyway. This also applies to other mentions of *.bashrc* in this presentation.

9 Searching the Past

- There are several ways of finding previous lines from history

10 Searching the Past

- There are several bad ways of finding previous lines from history
- Many people go for pressing Up lots (and lots)
 - A tad inefficient, perhaps
- Canner folk know that Ctrl+R searches previous lines
 - But Ctrl+R zip Esc doesn't find the last zip command — it also matches any line that copied, deleted, unzipped, or did anything else with a zip file
- Those of a gambling bent can chance ! and a command name
 - Irritating when !gv opens gvim instead of gv

11 Sane Incremental Searching

- Bash can cycle through lines starting in a particular way
- Just type in a few characters then press Up
 - Don't need to press Up so many times
 - Don't see lines that merely contain those letters
 - Don't have to chance executing the wrong line

zip Up goes to the most recent zip command. Further presses of Up cycle through previous zip commands.

gv Up goes to the most recent command starting with those letters. Suppose that found a gvim command and you

were looking for a `gv` command: pressing `Space Up` would then go to the most recent line starting `gv_`.

12 Configuring Up and Down

- Incremental searching with `Up` and `Down` is configured in `.inputrc`

```
"\e[A": history-search-backward
"\e[B": history-search-forward
```

- Old behaviour still available with `Ctrl+P` and `Ctrl+N`
- If that prevents `Left` and `Right` from working, fix them like this:

```
"\e[C": forward-char
"\e[D": backward-char
```

13 Repeating Command Bits

- Commonly want to repeat just bits of commands
- Very often the previous command's last argument
- `Meta+.` types this
 - Press repeatedly to cycle through the final argument from earlier commands

Suppose these commands have been executed:

```
$ mount /mnt/cdrom
$ ls /var/tmp
```

It is then possible to type the start of the next command line (such as `ls -l`) and simply press `Meta+.` to have `/var/tmp` be 'typed' for you. Pressing `Meta+.` again (without an intervening keystroke) will 'untype' `/var/tmp` and replace it with `/mnt/cdrom`.

Note that on PCs `Alt` typically functions as `Meta`, so `Alt+.` is what would be pressed. This applies to all mentions of `Meta` in this talk.

14 Grabbing Other Bits

- Other bits of previous commands can be grabbed with `!`
- `!:0` is the previous command name
- `!^`, `!:2`, `!:3`, ..., `!$` are the arguments
- `!*` is all the arguments
- `!-2`, `!-3`, ... are earlier commands
 - `!-2^`, `!-2:2`, `!-2$`, `!-2*`
- So can have things like

```
$ !-2:0 -R !^ !-3:2
```

- But looks like nonsense
- A brave person who presses `Enter`

15 Magic Space

- A magic space inserts a space character as normal
- And also performs history expansion in the line
- See what you type before you commit to it
 - Press `Space` before `Enter` if necessary

Suppose you squeeze a png image to use maximum compression:

```
$ pngcrush pineapple.png crushed_pineapple.png
```

You can then list the files' sizes without having to type their names again:

```
$ ls -lh !*
```

Then you can rename the new file to the original name (deleting the uncrushed file in the process). The new name can be 'typed' with `Meta+.`, and the original name picked out of a previous command:

```
$ mv Meta+. !-2^
```

Before committing a file to CVS you check over which changes you've made. Vim conveniently uses colour to highlight the changes, and using a separate window means that they can be kept on-screen while typing the commit message:

```
$ cvs diff GBdirect/DocTools/Util.pm | gview -
```

Then the file can be committed, picking its name out of the previous command line:

```
$ cvs com !:2
```

16 Magic Space Set-Up

- Magic space is configured in `.inputrc`
- Redefine what `Space` does
- There are other readline-based programs without this feature, so make it only apply in Bash:

```
$if Bash
  Space: magic-space
$endif
```

For example the MySQL client and the Perl debugger can also use the readline library.

17 Forgetting Options

- Common to forget an option from a command
- Want to rerun the command with the option
- Go to the previous history line, then move just after the command name to type the option
- Can set up a keyboard macro to do this

If you (attempt to) view a directory's contents:

```
$ ls -l /var/spool/exim/
ls: /var/spool/exim: Permission denied
```

then to see the permissions of the directory itself, add the `-d` flag:

```
$ ls -d -l /var/spool/exim/
```

`grep` can search through all files in a directory:

```
$ grep RewriteCond /usr/share/doc/apache/manual/
```

```
grep: /usr/share/doc/apache/manual: Is a directory
```

But only if you remember the `-r` flag:

```
$ grep -r RewriteCond /usr/share/doc/apache/manual/
```

Sometimes processes refuse to die:

```
$ killall xms
```

The `-9` flag leaves them with little choice:

```
$ killall -9 xms
```

Other places where flags can be added include `ls -tr`, `rm -r`, and `chmod -R`.

18 Insert-Option Macro

- `Meta+O` can be made to load the previous command and position the cursor for typing an option

- Defined in `.inputrc`:

```
"\M-o": "\C-p\C-a\M-f "
```

- `Ctrl+P`: previous line
- `Ctrl+A`: start of line
- `Meta+F`: forward a word, past the command
- `␣`: insert a space

- 17 unused keystrokes with just `Ctrl` or `Meta` modifiers

The 2 unused keystrokes with `Ctrl` are the rather awkward `Ctrl+\` and `Ctrl+^`.

But there are 15 letters available for use with `Meta`, namely: `Meta+A`, `Meta+E`, `Meta+G`, `Meta+H`, `Meta+I`, `Meta+J`, `Meta+K`, `Meta+M`, `Meta+O`, `Meta+Q`, `Meta+S`, `Meta+V`, `Meta+W`, `Meta+X`, and `Meta+Z`.

19 Default Command Options

- One way to avoid forgetting common options is to have Bash remember them
- Define functions to execute commands with desired options
- Name the functions to replace commands

`ls` can be made always to include the `-F` flag, to add symbols marking directory and command names among the list of filenames.

`mkdir` can have the `-p` flag so that it is possible to create nested subdirectories in one go.

`grep` can have the `-r` flag so that it will work on directories. This won't cause any harm when grepping ordinary files.

Other default flags to consider include `scp -pr`, `dirs -v` and `jobs -l`.

20 Defining Functions

- Define functions providing default options in `.bashrc`

```
function ls
{
  command ls -F "$@"
}
```

- `command` runs the real command
- `"$@"` inserts the user arguments

- Do not define `rm` to include `-i` by default

- Won't be there on other people's systems

21 New Command Names

- Commands with arguments can have different names:

```
function duff
{
  diff -ur "$@"
}
```

- Safe to export, for use in shells embedded in editors:

```
export -f duff
```



- Leaves the original name alone for programs relying on it

This `ll` function produces a long-format directory listing:

```
function ll
{
  command ls -Flh "$@"
}
export -f ll
```

`doc` can be made to change to the specified package's documentation directory and display the files therein:

```
function doc
{
  pushd "/usr/share/doc/$1" && ls
```

```
}  
export -f doc
```

Functions can be created for performing file conversions. For example this function takes a single XFig file, such as *network.fig*, and produces a PostScript file with the same basename, such as *network.ps*:

```
function fig2ps  
{  
    fig2dev -L ps "$1" "${1%.fig}.ps"  
}  
export -f fig2ps
```

22 Graphical Commands

- Functions can also be used to ensure that graphical commands always open in the background:

```
function gimp  
{  
    command gimp "$@" &  
}
```



- Saves having to type the `&` every time

This is useful for pretty much any command which opens a window, including `gv`, `mozilla`, `acroread`, `xfig`, and `ooffice`.

23 Specifying Directories

- Bash can help with specifying directory names
- `cd -` changes back to the previous directory
 - Handy if you forgot to `pushd`
- In general `~-` is the previous directory
 - Useful for working with files in 2 different directories

If you have a tarball in the current directory but you wish to extract it in a different directory, you can change directory then use `~-` to refer to the directory containing the tarball:

```
~/downloads/Mozilla/ $ cd /var/tmp  
/var/tmp $ tar xzf ~-/moz Tab Meta+S Meta+S
```

Afterwards you can change back to where you were:

```
/var/tmp $ cd -  
~/downloads/Mozilla/ $
```

24 Directory Name Typos

- When changing directory small typos can be ignored by Bash
- Enable this in `.bashrc`:

```
shopt -s cdspell
```

Bash will cope with each component of the typed path having one missing character, one extra character, or a pair of characters transposed:

```
$ cd /vr/lgo/apaache
/var/log/apache
```

25 Directory Bookmarks

- Some directories are changed to more frequently than others
- Can avoid typing their full paths if their *parents* are in `$CDPATH`

Suppose `$CDPATH` contains `~/pending` and `/home/www-data`; you would then be able to change to their subdirectories from anywhere on the system without typing a full path:

```
$ cd conference
/home/simon/pending/conference
$ cd intranet/logs
/home/www-data/intranet/logs
```

If `..` is also in `$CDPATH` then you can easily change to sibling directories. For example, following on from above you could do:

```
$ cd docs
/home/www-data/intranet/docs
```

26 Setting `$CDPATH`

- Set `$CDPATH` in `.bashrc`:

```
CDPATH=' .:~:~/links:~/~/projects:/var/www/virtual_hosts'
```

 - Colon-separated
- Put `.` first so can still change directory normally
- Also could include:
 - `..` for sibling directories
 - `../..` for aunts and uncles
 - Home directory
 - A directory just containing links to often-used directories

27 Completion

- The way Bash completes filenames and other things with Tab can be customized in *.inputrc*

- Keep hidden files hidden when doing, for example, `cp ~/` Tab:

```
set match-hidden-files off
```

- `cp ~/.` Tab will match hidden files

- Complete things that have been typed in the wrong case:

```
set completion-ignore-case on
```

- When listing possible file completions, put `/` after directory names and `*` after programs:

```
set visible-stats on
```

28 Completion Cycling

- Instead of beeping on an ambiguous completion request, Bash can be made to list the possibilities

- Then a keystroke can be used to cycle through them

- Often faster than working out which characters to type to be unambiguous

For example, suppose you want to edit your *.bashrc*. You could start by typing:

```
$ xemacs ~/.b
```

Then pressing Tab yields a list of matching files, and completes the name as far as possible:

```
.bash_history .bash_profile .bashrc
```

```
$ xemacs ~/.bash
```

Meta+S will cycle through the completions, so pressing it once gives:

```
$ xemacs ~/.bash_history
```

Tapping it another couple of times completes to the required filename.

29 Cycling Set-Up

- Cycling through potential completions is configured in *.inputrc*

- List the possible completions when Tab is pressed:

```
set show-all-if-ambiguous on
```

- Make Meta+S cycle through the list:

```
"\M-s": menu-complete
```

30 Programmable Completion

- Bash's programmable completion enables commands' arguments to be completed intelligently for different commands

- Enable it from *.bashrc*:

```
source /etc/bash_completion
```

When writing a document you often end up with several files with similar names but different extensions:

```
$ ls
Bash_tips.aux Bash_tips.log Bash_tips.pdf Bash_tips.tex
```

But with programmable completion, Bash will pick the filetype that matches the command and ignore all the others:

```
$ acroread ba
```

Pressing `Tab` converts the above to:

```
$ acroread Bash_tips.pdf
```

If that was the only PDF file in the directory you don't need to type *any* of the filename: just type the command name then press `Tab` and the filename will be inserted. And even if there are a few PDF files, it still may be quicker not to type any of their names and just use `Meta+S` to cycle through them.

This is useful for any application which only works with a limited group of filetypes. It can also be used to exclude filetypes from commands. For example image and sound files can be excluded from filename completions for text editors.

31 Completion Caveats

- Programmable completion doesn't always do what you want
- Sometimes worse than the default
 - Can be disabled for a particular command

```
complete -r cd
```
- Developed independently of Bash
- Download the latest version from <http://www.caliban.org/bash/#completion>

32 Accidental Exiting

- `Ctrl+D` conveniently exits Bash
- Sometimes *too* conveniently
- Specify that it must be pressed twice to exit:

```
export IGNOREEOF=1
```

```
$ Ctrl+D
$ Use "exit" to leave the shell.
$ Ctrl+D
$ exit
```

33 Summary

- Many things can be done to make using the command line more comfortable
- Main ones for me are keeping all history, searching backwards with `Up`, and cycling completions with `Meta+S`
- Downside of such luxuries is that that sometimes you have to use systems where they aren't configured
- Would fix the problem if everybody standardized on the settings presented here. . .

