Introduction

The *Linux Supporters Group* (LSG) aims to share experience and advice to solve computer problems.

One such problem occurs for `Windows` refugees who dual-boot the `debian`, `slackware`, `ubuntu` &c. variants of `Linux`, namely: that desktop menus, while useful, are nowhere near sufficient for doing some real work – i.e., solving problems for which *no menu item exists*. As you become familiar with the command-line, you'll find that this extends to cover many tasks that you'll want to do.

asm — basic — C

You can do anything at all on a PC using `assembler`. Of course `basic` was invented to avoid assembler. Clearly `C` was invented to avoid basic. It is obvious that the `shell` was invented to avoid programming in C. Some even believe that the `GUI` was invented to avoid the shell ...

cmd — psh — bash

Users of `MS-DOS` (command.com) have a very primitive method for doing things, and historically thousands of them have routinely turned to augmenting that operating system by programming in `8086 assembler`, `basic` and/or in `C`. Countless improvements and additions to `DOS` were circulated amongst enthusiasts between about 1985 and 1995, without any official support: `FreeDOS`, `4DOS`, `4NT`, `the Hamilton-CShell`, – most spectacularly `Linux` and `bash` – were all born through this movement. The `NT` series introduced some improvement with `cmd.exe`, but it remains awkward to use. The `Vista` distribution has been recently released, but without the promised new shell and scripting language known as `monad` (now `powershell`).

It is easy to install `cygwin` on an `XP` machine. This gives you the Linux CLI. In addition, `cmd.exe` runs natively on `XP`, and `powershell` v1.0 was released to try out on `XP` around November 2006.

What is a Shell ?

A shell is a program, running at a terminal, that executes commands that are typed in response to its prompt. The shell processes user commands and *scripts* which are stored sequences of previously-debugged commands. With the shell you can build up a wide range of your own re-useable applications with a meccano-like set of small tools.

It is a *whole lot easier* and much more productive to write simple tools using the command line than it is to wrestle with graphical toolkits.

So, what commands can I use?

When you start out using the command-line, it's all a bit of a mystery. You do not have any idea of what commands are available, how to implement them or what they can do to solve *your* particular problem.

Let's teach you how to fish

This talk compares the three shells `cmd`, `psh` and `bash` by constructing a simple application using each shell that allows you to explore the various available commands at random. This way you can discover new commands that you did not know were possible or available on your system. Once you know these available commands you can start exploring the command line for yourself.

Browsing the manuals of the available commands

The first thing we do is to make a list of all the commands; builtin shell commands and scripts and executables. Once we have a list of all the commands, we can get the help file or the manual for each command, and view the first few lines of it. We shall now implement this plan in each shell.

CMD: complete script to browse manuals

You can normally get help on a program in NT by typing *(e.g.)* `help cmd`. Many programs don't use this, instead needing the form *(e.g.)* `cipher /?` or *(e.g.)* `arp -h`. Many have no help at all ...

```
 1  :: explore.bat     usage: explore.bat [numberofscreenlines]
 2  echo off & setlocal enableextensions enabledelayedexpansion
 3  if [%1]==[] (set /a lines=20) else (set /a lines=%1)

 4  echo this list of executables is from C:\WINDOWS\system32>exe.lst
 5  echo for getting help on 'prog.exe', try typing 'prog /?'>>exe.lst
 6  echo or use 'windows server 2003 resource kit tools help'>>exe.lst
 7  dir /B C:\WINDOWS\system32|find ".exe">>exe.lst  &  more exe.lst

 8  help|find /v "For more info"|find /v "        ">help.tmp
 9  help|more  &  del help.ran>nul
10  for /f "tokens=1" %%c in ('type help.tmp') do (
11   echo !RANDOM! %%c>>help.ran )
12  type help.ran|sort>help.lst

13  for /f "tokens=2" %%c in ('type help.lst') do (
14   cls & help %%c>man & if exist man call :peruse
15   echo. & set /p answer="q=quit, enter=next< "
16   if /i "!answer!"=="q" (endlocal & goto :EOF) )
17  goto :EOF

18  :peruse
19  set /a count=%lines%
20  for /f "tokens=*" %%m in ('type man') do (
21    echo %%m & set /a count=!count!-1
22    if "!count!"=="0" goto :EOF)
```

CMD: explanation of script

This DOS batch file is run by setting an alias pointing to the script, and then invoking it:

```
cmd> doskey explore="%HOMEPATH%\explore.bat" $*  &  explore 25
```

line 1 is a comment (`::`) with the program name and how to use it.

lines 2 and 3 set necessary options for processing this script. They default the display lines to 20 unless this is specified on the command line.

lines 4 to 7 prepare a listing of the executable programs usually found in `C:\WINDOWS\system32`. Many of these have no on-line help, or fail to respond properly to the usual help commands (`prog.exe /?` or `prog.exe -h`) so we just get a list to look at instead of any actual help.

lines 4, 5 and 6 are just useful information for the heading of this list.

line 7 finds all the `.exe` files in the system directory, list only their names (`/B`) and add them all to the temporary file (`>>exe.lst`), then views the list on the screen using the `more` pager.

lines 8 and 9: In `cmd.exe` a list of all the system commands can be constructed by getting the normal help listing (`help`) which shows all the system commands and a synopsis of each one (try it); we do not use the first line (`find /v "For more info"`) nor any of the continued lines starting with 8 spaces (`find /v "        "`). We also have look at the `help` synopsis.

lines 10 to 12 make a list of commands in random order. Take each line of the help file we saved (`type help.tmp`) and select the first field (`"tokens=1"`) of each line, which is put into a variable called `c`; generate the next random integer (`!RANDOM!`) and write that integer and the command name (`%%c`) both onto the next line of `help.ran`. Then put the commands into a random order by sorting on the random integers (`sort`).

lines 13 and 14 take each line of the randomised help file we saved (`type help.lst`) and select the second field (`"tokens=2"`) of each line (the command name) which is put into a variable called `c`. The screen is cleared and the help information for that command is redirected to a temporary file (`help %%c>man`); then the desired number of lines of that help is printed on the screen by the subroutine (`call :peruse`);

line 15 prompts you for a decision (`set /p answer="q=quit, enter=next< "`) and then reads your response into a variable called `answer`.

line 16 quits this script if `q` or `Q` was pressed (`if /i "!answer!"=="q"`) or returns to the next command in the `for` loop otherwise. (Note that `enabledelayedexpansion` with the variable in the form `!answer!` rather than the normal `%answer%`, is crucial for getting this to work).

lines 18 to 22 are the `peruse` routine which prints `%lines%` of file `man` by counting down to zero.

PSH: complete script to browse manuals

Note that you can get help on a `topic` in `psh` by typing at the prompt (replacing `topic` with the command name or a guess at it): `psh> get-help topic -full|more` (but I warn against `more`). For now, in your home directory, open up a notepad file called `explore.ps1` and enter this text:

```
# explore.ps1    make executable:  psh> set-executionpolicy remotesigned
# make alias:    psh> function explore($n) {& $HOME\explore.ps1 $n}
$n=[int]$args[0]
$lines=20; if ($n -gt 0) {$lines=$n}
get-childitem $env:WINDIR\system32\*.exe|select-object name|C:\WINDOWS\system32\more

get-command|select-object name|tee command.names|C:\WINDOWS\system32\more

$random=new-object random
if (test-path -type leaf -path nnp.tmp) {remove-item nnp.tmp}
foreach ($line in $(get-content command.names)) {
add-content -path nnp.tmp -value "$($random.next()) $line" }
get-content nnp.tmp|sort|set-content -path sort.tmp

if (test-path -type leaf -path names.tmp) {remove-item names.tmp}
foreach ($line in $(get-content sort.tmp)) {
  $name="$line".split(' ',[stringsplitoptions]::removeemptyentries)|‘
    select-object -first 2|select-object -last 1
  add-content -path names.tmp -value "$name" }

$COMMANDS=$(get-content names.tmp|
  where-object {$_ -notmatch "Name"}|where-object {$_  -notmatch "----"})
foreach ($command in $COMMANDS) {
  get-help $command > man.tmp
  if (test-path -type leaf -path man.see) {remove-item man.see}
  $MANUAL=$(get-content man.tmp|where-object {$_ -notmatch "^[ ‘t]*$"})
  foreach ($line in $MANUAL) {"$line" >> man.see}
  clear-host
  if (test-path -path man.see -type leaf) {
    get-content man.see|select-object -first $lines}
  write-host ""; $ans=read-host "q=quit, enter=next< "
  if ("$ans" -eq "q") {
    $n=$(get-content command.names).length
    write-host "   found $n powershell cmdlets";
    remove-item command.names,sort.tmp,nnp.tmp,names.tmp,man.tmp,man.see
    exit 1}
}
exit 0
```

PSH: explanation of script

lines 1 and 2 are comments (`#`) with the name of the script (`explore.ps1`) and some usage information: how to point to the script and then execute it.

lines 3 and 4 set the number of screen display lines, if you put it on the command line by typing, for example, `explore 30` (but you may leave it off and it defaults to 20).

line 5 lets you see all the executables. It makes a list of the commands by searching the directories that contain the commands. In `psh.exe` executables are mostly in `$env:WINDIR\system32`.

line 7 gets a list of all the possible shell commands or *commandlets* (`get-command`); out of the listing we select only the name of the command (`select-object name`) and save all these names into a temporary file (`tee command.names`) before viewing on-screen.

lines 8 to 11 prepare to generate random numbers, like this: create a random-number-generator by making a new object of that type (`$random`); delete any previous file named `nnp.tmp`; for each command name on a line in our list of commands generate a random number to tag it with (`$random.next()`) and add it, with the name of the command, as a number+name pair (e.g. `2987647 Set-Content`) to a file called `nnp.tmp`.

line 12 puts those command names into random order, by sorting the file `nnp.tmp` on the random numbers, thus sorting the commands into random order, and saves in `sort.tmp`;

lines 13 to 17 remove any existing file `names.tmp`; for each line in our sorted file, consisting of a number and a command name (such as `2987647 Set-Content`), split the line in two, discard the first part (the number) and select the second part (the name) and add that name to the temporary file `names.tmp`.

lines 18 and 19 get a list of the commands. The file `names.tmp` has a heading and a line (`Name` and `----`) which we need to ignore while making a list of the commands in random order. This list is then saved in a shell variable called `$COMMANDS`.

lines 20 starts looping through every `$command` in our `$COMMANDS` list.

line 21 puts the help information for the command into a file (`man.tmp`).

lines 22 and 23 strip out blank lines from the manual listing to compact it: (`-notmatch "^[ ‘t]*$"`) meaning reject lines that start (`^`) but then only have spaces or tabs of any number (`[ ‘t]*`) before the end of the line (`$`);

line 24 adds all such stripped-down lines to the `man.see` file.

lines 25 to 27 clear the screen (`clear-host`); if the manual exists then the first `$lines` are selected and displayed on the screen.

lines 28-33 give a choice to continue to the next command by prompting for a response into a variable `ans` (`$ans=read-host "q=quit...`); if that response is `q` then it quits after cleaning up; otherwise it goes on to the next manual (the last `}` on line 34).

BASH: complete script to browse manuals

You can get help on any *topic* in `bash` by typing at the prompt: `man topic` (or if you have a guess at the keyword) `man -k topic`. Save the following script as `explore` in your home directory:

```
#!/bin/bash
# explore manuals for linux commands in random order so discover new ones
# setup: chmod a+rx explore; echo "alias explore='$HOME/explore'">>.bashrc
# setup: source .bashrc     # to invoke:  explore  OR(eg)  explore 40

if [ -z "$1" ]; then let lines=20; else let lines=$1; fi

directories="/bin /sbin /usr/bin /usr/sbin /usr/local/bin"
commands=$(/bin/ls $directories|grep -v /)
commands="$commands builtins"

echo $commands \
 |tr ' \t' '\n' \
 |grep -v "^[ \t]*$" \
 |sort -u \
 |tee explore.commands \
 |tr '\n' ' ' \
 |fold -s -w 80 \
 |less

rm -f explore.random
for command in $(cat explore.commands)
do
  echo "$RANDOM $command">>explore.random
done

commands=$(cat explore.random|sort -n|awk '{print $2}')
for prog in $commands
do
  clear
  man $prog|col -b|grep -v "^[ \t]*$"|head -$lines
  echo; read -p "q=quit, enter=next< " ans
  if [ "$ans" == "q" ]; then break; fi
done

echo "found $(cat explore.commands|wc -w) user commands"
rm explore.commands explore.random
exit 0
```

BASH: explanation of script

line 1 just says to use the program `/bin/bash` as the interpreter for this script.

lines 2,3 and 4 are comments starting with `#`, showing usage information. The `echo` adds the alias to `.bashrc` to ensure that it is re-established every time you open a terminal.

line 5 sets the number of display lines if it is specified on the command line, otherwise it defaults to 20 lines.

line 6 defines a list of directories that contain commands. In `Linux` these directories are generally: `/bin` for normal binaries; `/sbin` for system binaries; `/usr/bin /usr/sbin /usr/local/bin`.

lines 7 and 8 make a list of files that are in these directories, and store that list in a shell variable named `commands` for use, as well as adding the `builtins` manual page to the list.

lines 9 to 16 look at them all by running the commands though a pipeline. In English, this means: list our commands on the screen (`echo $commands`) one per line by converting spaces and tabs to newlines (`tr ' \t' '\n'`) and ignoring any blank lines (`grep -v "^[ \t]*$"`) (lines that start and have only spaces and tabs before ending), so we can get them into alphabetical order of unique commands for viewing (`sort -u`); save them for future manipulation in a temporary file (`tee explore.commands`) then change all the line feeds (`\n`) back to spaces (`' '`) so the commands appear on one long line (`tr '\n' ' '`); format this line (`fold`) so that it breaks the line and starts a new one but only where there is an existing space (`-s`), and let it go across the screen for a width of 80 columns (`-w 80`); and let us see them all by pausing inside a paging program (`less`).

All that done in one line! — Well, was that like pulling teeth, or what?

lines 17 to 21 generate and store random numbers. There are over 1800 user commands in a typical `Linux` distribution. From the point of view of discovering available commands on your own, it is better to peruse the commands at random, so that every time you explore the commands different ones come up for consideration. This way you can become familiar with new possibilities. In English: remove any existing random file; for every command in our list of commands, generate a new random number (`$RANDOM`, which is an environmental variable) and add that random number and the command name to a temporary file (`>>explore.random`).

line 22 replaces our list of commands with this list, ordered numerically (`sort -n`, which effectively makes a random list of commands) and then selecting the 2nd field of each record (`awk '{print $2}'`) ignoring the 1st field, which is the number used to randomise it.

lines 23 to 29 browse the first `$lines` of the manual for every command, which should be enough to give you a good idea of the command, which means: for every single program (`prog`) that is in our list of programs (`$commands`) do the following set of operations (between the `do` and the `done`); clear the screen to give us a clean pallette (`clear`); look at the manual for that program (`man $prog`); remove any formatting commands in the manual (`col -b`) and compact it a bit by ignoring any blank lines (`grep -v "^[ \t]*$"`) (lines that start and have only spaces and tabs before ending), but only so many as fit on a screen easily (`head -$lines`).

line 27 when the first `$lines` of that manual are on the screen, ask us if we want to bomb out or look at the next command manual; then read our typed response into a variable (`ans`).

line 28 checks if the response is 'q' (`if [ "$ans" == "q" ]`) then (`then`) break out of this do-done loop (`break`) to line 30 and thus exit the program; otherwise just go on to the next command in the loop by ending the 'if' condition (`fi`) and complete the do-done loop (`done`).

lines 30 to 32 summarise the number of commands that were found on the system by doing a word count (`wc -w`) on the command list (`explore.commands`) before cleaning up temporary files and exit.

Comparison of Shells

Make sure you look for and utilise all or most of these features (in alphabetical order):

– Aliases – Case conditional branching – Command chaining – Command-line expansion – Conditional execution of subsequent commands – Conditional If-Then-Else – Error return codes – History – Job control Foreground/Background processing; – Loops For-While-Until – Menu creation – Positional parameters – Quoting and Escaping – Redirection – Sequential execution via pipes – Script-capture for analysis.

Refer to the description of each shell script (preceeding pages) to see how you can get help on a `topic` in these shells by typing `topic` or a guess at it at the command prompt.

parameter specification is very similar in all shells.

```
cmd>  %1 %2           [but you need to use %%1 %%2 in a batch file]
psh>  $1 $2 ... $9    [but there's no 'shift' command! - what were they thinking?]
bash>  $1 $2 ... $9   [and you use 'shift' to feed in more variables > 9]
```

variable allocation has some tricks up its sleeve.

```
cmd>   set n="%1"   [generally]        set /a n=%1    [for numbers]
psh>   $n="$1"      [generally]        $n=[int]$1     [for numbers]
bash>  n="$1"       [generally]        let n=$1       [for numbers]
```

referring to variables is very similar in all shells.

```
cmd>   echo %filename%
psh>   out-host "$filename"
bash>  echo "$filename"
```

continuing command lines is very similar in all these shells.

```
cmd>   type file.txt | ^          [the circumflex ^ continues lines]
psh>   get-content file.txt | `   [the backtick ` continues lines]
bash>  cat file.txt | \           [the backslash \ continues lines]
```

conditional execution has some little wrinkles.

```
cmd>   if "%option%" == "test"   set /a x=3           [the /a is for arithmetic]
psh>   if ( "$option" -eq "test" ) { $x=3 }           [more like C# program syntax]
bash>  if [ "$option" == "test" ]; then let x=3; fi   [if-then-fi delimits action]
```

executing a chain of commands is very similar in all these shells.

```
cmd>   now & freq 15 big.txt & now      [the & separates subsequent commands]
psh>   now; freq 15 big.txt; now        [the ; separates subsequent commands]
bash>  date; freq 15 big.txt; date      [the ; separates subsequent commands]
```

defining aliases is quite different in each shell.

```
cmd>   doskey freq="%HOMEPATH%\freq.bat" $*      [$* means all script parameters]
psh>   function freq($n,$text) { & "$HOME\freq.ps1" $n $text }  [note parameters]
bash>  alias freq='$HOME/freq'                   [requires no parameter spec.]
```

control loops have some slight differences in format.

```
cmd>   for %a in (%LIST%) do (
         call script.bat %a  )                    [note 'call' to invoke script]
psh>   foreach ($a in $LIST) {
         ./script.ps1 $a     }                    [note $a PERL heritage]
bash>  for f in $LIST; do ./script $f; done       [do-done delimits the action]
```

But watch out for these ...

multi-line command editting is very different. I consider it completely broken in `cmd` and `psh`.

```
cmd>   for %a in (%LIST%) do (
More?    call script.bat %a
More?  )                              [now press the UP ARROW to edit this]
cmd>   )                  [which is useless; it's only the last character typed]

psh>   foreach ($a) in ($LIST) {
>>       ./script.ps1 $a
>>     }                              [now press the UP ARROW to edit this]
psh>   }                  [which is useless; it's only the last character typed]

bash>  for f in $LIST
>      do
>        ./script $f
>      done                           [now press the UP ARROW to edit this]
bash>  for f in $LIST; do ./script $f; done     [the whole command ready to edit]
```

when referring to multiple items be careful with the unusual comma-delimited list.

```
cmd>  del a b c                    [works fine]
cmd>  type a b c>x                 [works fine]
psh>  remove-item a,b,c            [you need commas, not spaces]
psh>  cat a,b,c >x                 [you need the space in 'c >']
bash> rm a b c                     [works fine]
bash> cat a b c>x                  [works fine]
```

getting user input from the terminal has some idiosyncracies.

```
cmd>   set /p file="enter filename< "              [gets a file name from user]
psh>   $file=read-host -prompt "enter filename< "  [gets a file name from user]
bash>  read -p "enter filename < " file            [gets a file name from user]
```

paging a file within a script in `psh` fails to return when you quit using `out-host -paging` or `more`; both are completely broken — you need the older `C:\WINDOWS\system32\more` instead!

```
psh>  get-content $file|C:\windows\system32\more    NOT: get-content $file|more
```

timing a Command: if you want to time how long a script takes, `bash` does that *and* produces the normal result of the script, but `psh` fails to produce the result of running the script at all!

```
cmd>  now & script.bat & now        [works fine, but calculate time yourself]
psh>  measure-command {./script}    [produces NO OUTPUT from the script]
bash> time ./script                 [normal output plus execution time]
```

Further Assistance

There are some great books in the computer section of certain bookshops, and also some web sites that I can thoroughly recommend.

for learning bash a very good book at every level of expertise is *A Practical Guide to Linux: Commands, Editors and Shell Programming* by Mark Sobell.

also for bash there is a great tutorial by Mendel Cooper called *Advanced Bash-Scripting Guide: An in-depth exploration of the art of shell scripting* at `http://www.faq.org/faqs/`

to blow your mind Rob Flickenger's *Linux Server Hacks: 100 Industrial-Strength Tips & Tricks*

for learning powershell the best book I have come across that gets you up to speed straight away without any crap is *Professional Windows Powershell* by Andrew Watt.

and for cmd – well, best of luck! The FAQ file compiled by T. Salmi called `tscmd.zip` is listed near the bottom of `http://garbo.uwasa.fi/win95/dosshell.html` and is magnificent; and the *Server 2003 Resources Kit* contains a good reference manual in GUI form.
