Shell Scoops

John Weatherwax

February 4, 2006

1 Introduction

These are some note that I started making as I researched how to do various things in the shell. The idea for putting these items in a format like this came from the excellent book: UNIX Power Tools by Jerry Peek, Tim O'Reilly, and Mike Loukides [1]. Currently this writeup is more incomplete than I would like, basically containing simple instructions on only a few topics. As I get more time I plan on further developing these notes by including many more sections and topics and further expanding the individual sections into chapters.

2 Echoing Dots For Long Running Processes

Sometimes a shell script can take a long time to finish. Sometimes users would like a way to determine that the program is running. One common approach is to echo "dots" as the program executes. One method to do this is the following bit of code

```
while :; do printf "."; sleep 1; done &  ## or echo -n "."; or echo ".\c"
dotpid=$!
##  do your other stuff here
kill $dotpid
```

this doesn't handle things like killing the script before the sleep is finished,

3 Turning off the Console Beep

People often complain about the beep that accompanies filename completion. especially if you work around people who are sensitive to noise. To turn this feature off there seem to

be several proposed methods that may work depending on your system. Experiment to see which method does exactly what you are looking for.

- To turn this off for every user on your system, add this line to your /etc/profile

  ```
  echo -e \\33[10;50]\\33[11;1]
  ```

  I should point out that the line I've given (echo -e [blah...]) for the /etc/profile will do the job of just stopping the beep for filename completion, but has the useful advantage of not disabling the beep for other events - I find the beep useful for some things, like when I recieve new mail for instance.

- In the new GNOME window manager the terminals themselves have a setting to turn of the beep. To disable the bell, go to settings, then preferences, and then check the box that says silence terminal bell.

- There is also another way to do this: again in /etc/profile, put this:

  ```
  setterm -blength 0
  setterm -store
  ```

  This sets the bell length to 0.

- A simple "set bell-style visible" or "set bell-style none" in your .inputrc should be sufficient to silence the shell. To change the behaviour of bash when it cannot determine how to complete a filename, put either "set show-all-if-ambiguous On" or "set disable-completion On" in your ".inputrc"-file. This also has the side effect of preventing the beep.

- Another method that is X specific is given by

  ```
  xset b off
  ```

  (or instead of "off" use a number between 0 and 100 - which will set the bell volume to a percentage of its maximum)

  Again, I offer the xset b solution - I think this is the easiest of the solutions mentioned here - it won't fix the beep on startup, but when you are running your xserver, xset b will set the bell however you like, including pitch and duration (although I understand it may not work with all hardware). Here's an excerpt from the man page for xset:

```
xset  [-display  display]  [-b] [b on/off] [b [volume [pitch
     [duration]]]  .....
.
.
.
     b       The  b  option controls bell volume, pitch and dura-
             tion.  This option accepts  up  to  three  numerical
```

```
parameters, a preceding dash(-), or a 'on/off' flag.
If no parameters are given, or the 'on' flag is
used, the system defaults will be used. If the dash
or 'off' are given, the bell will be turned off. If
only one numerical parameter is given, the bell vol-
ume will be set to that value, as a percentage of
its maximum. Likewise, the second numerical parame-
ter specifies the bell pitch, in hertz, and the
third numerical parameter specifies the duration in
milliseconds. Note that not all hardware can vary
the bell characteristics. The X server will set the
characteristics of the bell as closely as it can to
the user's specifications.
```

- In addtion much more information on this topic can be found at:
  file:/usr/doc/howto/html/mini/Visual-Bell-1.html

4 Killing a Process by its Name

Often one wants to just kill a process by giving its name on the command line. Now the following:

```
ps -ef | grep processname | awk '{ print $2 }'
```

will return the pid of the process in question, but it can also return two pids - the process you are really interested in and that from the grep command. One method to avoid this problem is to pipe the result through a "grep -v" such as:

```
kill -9 `ps -ef | grep processname | grep -v grep | awk '{ print $2 }'`
```

Using xargs this same command becomes:

```
ps -ef | grep processname | grep -v grep | awk '{ print $2 }' | xargs kill -9
```

If you would rather have awk issue the kill command the following will work:

```
ps -ef | grep processname | grep -v grep | awk 'print "kill -SIG" $2' | sh
```

In addition if you are only interested in the command name, one can save a couple of greps, and make things run a lot faster with the following

```
kill -9 `ps -e | awk '$4 == "processname" { print $1 }'`
```

Also note that the process name (the fourth column of "ps -e" output) is truncated to eight characters. One can also use pattern-matching instead of string equality for cases when the process name has been truncated. This gives the following:

```
/usr/bin/ps -e | awk '$4 ~ /pattern/ { print $1 }' | xargs kill
```

Encapsulated in a nice function one obtains:

```
#!/bin/sh
# kill the named process in $1
killproc()
{
  pid=`/usr/bin/ps -e |
       /usr/bin/grep $1 |
       /usr/bin/sed -e 's/^  *//' -e 's/ .*//'`
  [ "$pid" != "" ] && kill $pid
}
```

Encapsulated as a alias would look like the following:

```
alias rmp 'ps -ef | grep !* | grep -v grep'
```

This returns the entire ps string, the next piece of code returns only the process id

```
alias rmp 'ps -ef | grep !* | grep -v grep | cut -f2 -d" "'
```

5 Unscrambling Your Path

Through various means sometimes one's path gets extreamly repetative, storing duplicate entries of various directories. This is not only a pain visually to look at but it is also slower to find a given executable of interest since it has to repeatadly search the same directories. The following perl script (fixpath.pl) can be run and retuns your $PATH without any duplicate entries:

```
#!/usr/bin/perl
```

```
foreach (split(/:/, $ENV{"PATH"})) {
    push(@paths, $_) unless defined $paths{$_};
    $paths{$_} = 1;
}
print join(":", @paths), "\n";
```

I find that this helps to keep my path clean and unique. It makes dealing with the $PATH environmental variable much more easily.

6 Floating Point Operations in the Shell

If you've done much shell scripting you'll quickly find that there is a noticable absance of numerical ability. The following are some tricks you can use to get around this deficiency.

- In a POSIX shell (e.g. bash, ksh), there is built-in integer arithmeitc. For instance in bash one can do the following:

```
$ echo $(( 11235 / 81321 ))
0
$ echo $(( 23 ** 2 ))
529
```

- If you pipe your expression to dc, one can get integer arithmetic opperations. The following are a couple of examples:

```
$ # 10 * 3
$ echo 10 3 \* p | dc
30
$ # 10 / 3
$ echo 10 3 / p | dc
3
$ # square root of 9
$ echo 9vp | dc
3
```

- If you pipe your expression to bc, it will do *integer* additions, subtractions, multiplications, and divisions for you. For example:

```
$ # 3 * 5 + 4
$ echo "3 * 5 + 4" | bc
19
$ # 10 cubed
$ echo 10^3 | bc
```

```
1000
$ # integer square root of 9
$ echo sqrt\(9\) | bc
3
```

One has to be carefull however. The default intialization of bc does *integer* division. So the following will report:

```
$ echo "3 / 5" | bc
0
$ # integer square root of 10
$ echo sqrt\(10\) | bc
3
```

This could be remidied by setting the "scale" variable in bc, as follows:

```
$ echo 'scale = 10; 3 / 5' | bc
.6000000000
$ # square root of 10 with 10 decimals
$ echo 'scale = 10; sqrt(10)' | bc
3.1622776601
```

For shell scripting might be a bit of a pain to do at each instance. For GNU bc one can add the "-l" flag to bc in to link in the floating point library with a default scale of 20. For example:

```
$ echo "3 / 5" | bc -l
.60000000000000000000
```

A benift of the "-l" flag is that it also allows the log and some other math functions:

```
$ # log of 294
$ echo "l(294)" | bc -l
5.68357976733868161102
$ # The sine of ~pi/2
$ echo "s(1.5707950)" | bc -l
.99999999999911980765
$ # The bessel function of integer order 0 of 1.0.
$ echo "j(0,1.0)" | bc -l
.76519768655796655144
```

- AWK is another highly portable option:

```
$ awk 'BEGIN {print 3/5; exit}'
0.6
```

Chris F.A. Johnson, uses the following shell function to do calculations with awk, and stores the result in the variable "result".

```
calc()
{
result=‘echo "" | awk ’BEGIN { OFMT="%f"; print ’"$*"’}’‘
}
```

Its usage is like the following:

```
$ calc "log(294)"
$ echo $result
5.683580
$ calc "11235 / 81321"
$ echo $result
0.138156
```

The additional

```
echo ""
```

is needed because without it some version of awk will hang[1].

- Yet another (highly portable) solution is perl:

```
$ perl -le ’print 3/5’
0.6
```

One might argue that perl would be very slow if you have to do many calculations. This can be somewhat simplified if you can group all your floating point operations into one place. Then perl is called only once for all the opperations to be done. The following perl script (calc.pl) can be used to achomplish this task:

```
#!/usr/bin/perl
if (@ARGV) {
    print eval("@ARGV"), "\n";
} else {   while (defined($expr=<>)) { $_ = eval($expr); print "$_\n" }
}
```

Then one can do simple calculation from the shell:

```
$ calc.pl 4 / 9
0.444444444444444444
```

Or even much more complex calculations in a single invocation of perl

```
$ printf ’4/9\n 3/17\n 3/5\n 18/2\n 22/7’ | calc.pl
0.444444444444444444
0.176470588235294118
0.6
9
3.14285714285714286
```

[1] SunOS 4.1

This script can even be used interactivly. Such as

```
$ calc.pl
 4*5
 20
 $_ / 6
 3.33333333333333333
```

- In terms of size the ordering is

  ```
  shell < dc < bc < awk < perl
  ```

7 Warning: Missing charsets in String to FontSet conversion

This can happend because the $LANG variable is not set correctly. In this U.S.A. This can be fixed with the following:

- For a C shell, type

  ```
  setenv LANG en_US.iso88591
  ```

- For a Bourne or Korn shell, type

  ```
  LANG=en_US.iso88591; export LANG
  ```

8 Adding a Final Newline

Some (buggy) versions of the UNIX utilities require that the files they process end in a newline. If they don't often the last line of the file gets dropped. There is a version of sed that will perform this way. There are a couple of ways to insure that your files *have* newlines.

- For each file issue the following command:

  ```
  echo "" >> file
  ```

  The problem with this technique is that several files can then have more than one newline.

References

[1] S. Powers, J. Peek, T. O'Reilly, M. Loukides, et al. *Unix Power Tools*. Third edition, 2002.