Unix login Profile

A general discussion of shell processes, shell scripts, shell functions and aliases is a natural lead in for examining the characteristics of the login profile. The term "*shell*" is used to describe the command interpreter that a user runs to interact with the Unix operating system. When you login, a shell process is initiated for you, called your *login shell*. There are a number of "standard" command interpreters available on most Unix systems. On the UNF system, the default command interpreter is the Korn shell which is determined by the user's entry in the `/etc/passwd` file.

From within the login environment, the user can run Unix commands, which are just predefined processes, most of which are within the system directory named `/usr/bin`.

A *shell script* is just a file of commands, normally executed at startup for a shell process that was spawned to run the script. The contents of this file can just be ordinary commands as would be entered at the command prompt, but all standard command interpreters also support a *scripting language* to provide control flow and other capabilities analogous to those of high level languages.

A *shell function* is like a shell script in its use of commands and the scripting language, but it is maintained in the active shell, rather than in a file. The typically used definition syntax is:

```
<function-name> ( )
{
    <commands>
}
```

It is important to remember that a shell function only applies within the shell in which it is defined (not its children).

Functions are usually defined within a shell script, but may also be entered directly at the command prompt. For the Korn shell, entering either the function header or

```
function <function-name>
```

at the command prompt will initiate function entry for the current shell. The shell uses a "secondary prompt" rather than the command prompt during function entry. Function definition ends when the right brace (`}`) is entered at which point the shell returns to the regular command prompt.

Functions are executed like commands. Users typically customize their work environment by defining functions that are created at login by their login script.

Users can also customize their work environment by creating *aliases* to provide alternate syntax for commonly used commands; e.g.,

```
alias ll="ls -l"
```

defines `ll` to be an alias for the command string `ls -l`. Within the shell issuing the alias, `ll` can be used as if it were a command whose meaning is `ls -l`.

A list of all aliases in effect for a shell can be obtained by running

```
alias
```

without arguments.

1. `.profile` – the Korn Shell login script

At login, the system “forks” a new shell as the login shell for the user session. For the Korn shell (and the Bourne shell), if the user has a script file named "`.profile`" in the home directory, then it is run at startup to configure the login shell. The C shell has a similar prescriptive, except that the script file looked for must be named "`.login`".

Normally, at the time an account is created for a user, a basic `.profile` script file is installed in the home directory for the account, providing a default "standard" user environment. By editing this file, a user can customize his/her login environment.

A major purpose of the `.profile` script is to establish values of variables that configure the login session.

2. Shell variables

Variables are established within a shell simply by giving them values. Unless a variable has been made *readonly* using the `readonly` command, its value can be changed at any time by simply assigning a new value to it. A shell variable is *local*; i.e., changing its value does not affect the parent process. Assigned shell variables that have been *exported* by a parent process are *inherited* by all child processes. For an inherited variable, the initial value of the variable is obtained from the parent process.

The `export` command is used to place a shell variable on the shell's export list; e.g.,

```
export myvar
```

puts `myvar` on the export list for the shell; however, `myvar` cannot be inherited by child processes spawned by the shell until it has been assigned a value. For the Korn shell, the export list of the parent (assigned variables only) is inherited by the child (this is **not** true for the Bourne shell). Hence, if a spawned shell changes `myvar`, then any shell that it spawns will inherit `myvar` with the new value.

Running `export` without arguments will display the export list for the shell along with the values of those exported variables that have them (for the Bourne shell, the export list is not inherited; a variable is inherited instead from the parent that last exported it).

A *keyword variable* is one whose name has special meaning to the shell or to some command. By convention, keyword shell variable names are given in all capital letters; e.g., `PATH`. Some keyword variables are exported from outside the login shell and some are not (i.e., the user must provide for their export, if desired). Any keyword variable that has been exported will be inherited by child processes (with the parent's value of the variable).

There are some keyword variables (e.g., `PS1`, the keyword variable which controls the appearance of the command prompt) that are created with a default value if the variable is not inherited from the parent. In particular, for the Korn shell `PS1` will be created with the default value "`$ `" if it is not inherited otherwise. If `PS1` is exported, then it will be inherited thereafter, the disadvantage being that each spawned shell inherits its command prompt from its parent.

For the Korn shell, if an exclamation point ("`!`") is imbedded in `PS1`, then it will be expanded as the command number of the current command of the login session.

Examples are discussed later for producing a more informative prompt than "`$ `" for each active shell.

3. Variables and values

The current values of all variables that have been established within a shell can be determined by running the `set` command without arguments; i.e.,

```
set
```

lists all variables for the shell along with their values.

The `echo` command simply echoes its arguments to the display. In particular,

```
echo PATH
```

displays the text string "PATH". Although there is a keyword variable named PATH, in this case the argument that is passed to `echo` is just the name of the variable, not its value, so it is the name and not the value that is echoed..

`echo` is a useful means for determining how the shell will expand an entry that is not just simple text. The value of a variable is given by prefacing its name with $; i.e., $PATH represents the value of the variable named PATH. Hence,

```
echo $PATH
```

receives as its *argument list* the text string given by substituting the value of the PATH shell variable for $PATH and displays this list (compressing out extra spacing, among other things). To exhibit the actual string value of PATH (which may have extra spacing, for example), the entry

```
echo "$PATH"
```

is used (in this case `echo` receives only one argument, a single string whose value is the text content of the PATH variable).

A shell variable (unless it is `readonly`) can be removed from a shell simply by unsetting it; e.g.,

```
unset myvar
```

If `myvar` was on the `export` list, it is removed and will not be inherited by child processes unless exported again.

`unset` can also remove functions; e.g.,

```
unset -f myfun
```

removes the function `myfun` from the current shell.

4. Assigning values to variables

The syntax for assigning a value to a shell variable is

<variable-name>`=`*<value-to-assign>*

Note that there are no spaces around `=`. For the C shell, the `set` command must be used in assigning variables. All shell variables are *string variables* (i.e., the value is a character string).

The role of double quotes (`"`) and single quotes (`'`) can be confusing. For example, any one of

```
myvar1=mval
myvar1="mval"
```

and

```
myvar1='mval'
```

sets the value of `myvar1` to the string `"mval"`, since the command interpreter interprets each of `mval`, `"mval"`, and `'mval'` as the same string.

The command interpreter uses spaces to separate command line arguments, so quote marks must be used to assign a multiword string to a variable. Here, either

```
myvar2="mval more"
```

or

```
myvar2='mval more'
```

sets the value of `myvar2` to the string `"mval more"`. For this case both `"mval more"` and `'mval more'` are interpreted as the same string.

The difference between using double quotes and single quotes is that *variable substitution occurs inside double quotes*. Hence, for the shell variable `myvar2` from above, the value of `"$myvar2"` is `"mval more"`.

In contrast, the value of `'$myvar2'` is just `"$myvar2"` since variable substitution does not occur in this case. The difference can be easily checked by using `echo`; e.g.,

```
echo "$myvar2"
```

will display "*myval more*" whereas

```
echo '$myvar2'
```

will display just the string "*$myvar*".

5. Command substitution

In assigning values to variables, *command substitution* can be used to include the output of one or more commands in the value assignment. The original command substitution symbol is the accent mark (`), but the Korn shell also allows the form `$(`<*command*>`)`. If using (`), be careful not to confuse it with a quote mark. Within a double-quoted string, a command substitution `$(`<*command*>`)` takes the output of <*command*> and substitutes it for `$(`<*command*>`)`. For example,

```
demo="working directory: $(pwd)"
```

will substitute the output of the command `pwd` for `$(pwd)` in the string assigned to `demo`. Hence,

```
echo $demo
```

will output the string

`working directory:` <*working-directory*>

Variables are not expanded under command substitution; for example, the series of commands

```
varb='$PATH'
demo="$(echo $varb)"
echo $demo
```

will echo the string "`$PATH`" (i.e., the value of `$varb` is not expanded to be `$PATH` in `$(echo $varb)`).

For a slightly more complex illustration,

```
echo "user=$(whoami): $(pwd)"
```

will display the format

`user=`<*user's name*>`:` <*user's current working directory*>

6. Spawning a new shell

The `ksh` command is used to spawn (or fork) a new (Korn) shell. If no argument is provided, the input file is "`stdin`", which in effect means that the shell gets its input from the user. The `exit` command is used to return to the parent process, which is how the user ends input to the shell, in effect terminating the shell and returning to the parent that spawned it. The `sh` command spawns Bourne shells (`csh` spawns C shells). The alias

```
alias sh=ksh
```

will override this meaning, causing `sh` to be the same as `ksh`.

While in the spawned shell, commands that affect the shell environment (such as `cd`) do not affect the parent shell; i.e., if a spawned shell changes the working directory (which is inherited from the parent), when it terminates the parent shell will still be in the original working directory.

The command line

```
ksh myscript
```

forks a new shell which takes `myscript` as input, running the commands in the script. When finished, the system returns to the parent process (either by encountering an `exit` command in processing the script or by executing the last command in the script). The script may itself issue `ksh` commands, spawning shells for which it is parent.

If the file `myscript` has execute permission turned on (via `chmod`) and the first line of `myscript` is

```
#!/usr/bin/ksh
```

then it can be executed by simply entering

```
myscript
```

7. Running a script in the current process

Sometimes a user wants to run a script in the current shell, particularly to define functions or to set variable values in the current shell. This is done with the "dot" command. For example,

```
. myscript
```

runs the commands in `myscript` as if they were entered from the command line of the current shell (a new shell is not forked to run the script). The C shell uses the `source` command for this purpose.

Note that if `.profile` is run as a regular shell script using the `ksh` command, its variables disappear when the shell forked by `ksh` to run the script has finished. In contrast if `.profile` is run using

```
. .profile
```

then it will be run in the same manner as at login. Hence, if `.profile` is edited, then the changes can be implemented in this manner without having to log out and then log back in.

8. ENV - the environment variable for spawned shells

The Korn shell recognizes the *environment variable* `ENV` as a keyword variable. The value of `ENV` must be a script file, which is accordingly invoked at startup for any spawned shell (including the login shell). If `ENV` is set in `.profile`, then the script it points to will run as part of the login shell startup. The `ENV` script is typically used for variables, aliases, and function definitions that you want to establish in all spawned shells.

For example, suppose in your `.profile` script that you add the lines

```
ENV=~/.kshrc
export ENV
```

and that you have defined a file in your home directory named `.kshrc` which is a shell script with entries such as

```
alias sh=ksh
```

Then the alias for `sh` will remain in effect for the new shell.

The `ENV` name `.kshrc` used above is arbitrary, but is the common choice because it mnemonically mimics `.cshrc`, the name of the "***r**un **c**ommands*" script that the C shell looks for at startup.

9. The Unix `exec` command

When a user executes a command from the command line, the shell normally resumes when the command has finished. The dot command is used to run scripts in the current shell, which also leaves the current shell running. The `exec` command under the Korn shell is another story. In contrast to the dot command, the `exec` command *replaces* the current process (inheriting its characteristics). Hence, if a user runs

```
exec a.out
```

instead of the more usual

```
a.out
```

then the shell running the `exec` is replaced by the process `a.out`. When `a.out` finishes, control returns to the parent of the shell that ran the `exec`. Note that if the user was in the login shell, then the user is in effect logged out when the `exec` finishes.

Note that if `exec` is used in a script, no statements in the script below the `exec` will be run; i.e., the `exec` is the last thing the script runs so any trailing statements are meaningless. `exec` can also be used to run a script, but keep in mind that in this case, the shell running the script replaces its parent!

10. Some Examples

a) A variable name can be exported before it has any value, but it won't be inherited unless it has a value. For the command sequence

```
export new
ksh
export
```

the variable `new` will not appear in the export list for the spawned shell and will not be exported to any further shells spawned from this one.

In contrast, for the command sequence

```
export new
new=nvalue
ksh
export
```

the variable `new` will appear in the export list for the spawned shell with the inherited value "`nvalue`".

b) Commonly used system programs often recognize one or more keyword variables whose purpose is to allow users to establish an initial configuration for the program.

For example, initial configuration values for the `vi` text editor can be established by setting them in the `EXINIT` keyword variable. The variable is named `EXINIT` because `vi` was originally the visual interface for the line editor `ex`. Assigning a value to `EXINIT` is usually taken care of in the user's `.profile` login script.

For example, a typical `.profile` entry for `EXINIT` might be

```
EXINIT="set number showmode ignorecase
                        nomagic tabstop=2"
export EXINIT
```

It's important to `export` the keyword variable `EXINIT` so that the settings will be in effect for any spawned shell processes. The meaning of the settings can be found by consulting the "`man`" page for `vi`; i.e.,

```
man vi
```

c) For the Korn shell there are other keyword variables commonly established in `.profile`. In particular, a command "history list" of prior command entries is maintained by each Korn shell, the location of which can be controlled by the keyword variable `HISTFILE.` A "global" file for all shells can be used; e.g.

```
HISTFILE=~/.history
export HISTFILE
```

establishes that the file `.history` in the home directory is where the command history list is to be maintained by all shells.

The `fc` command (*fix command*) is used to access `HISTFILE` (most users employ the `history` command for this purpose, which is a preset `alias` for `fc -l`). The keyword variable `FCEDIT` is usually established in `.profile` to determine the default editor used by `fc` when it is invoked; e.g.,

```
FCEDIT=/usr/bin/vi
export FCEDIT
```

establishes `vi` as the default editor. `fc` can be used to fix and re-execute a sequence of commands, but is more

typically used in the `-l` (list) mode. Unless command numbers are specified in invoking `fc`,

```
fc -l
```

simply lists the last 16 commands executed.

d) For the Korn shell, the `set` command is used to establish usage norms for a user. A typical `ENV` entry is

```
set -o ignoreeof -o vi
```

`-o ignoreeof` establishes that an "EOF" (<Ctrl-D>) entered at the command prompt is ignored (otherwise, the EOF would terminate the shell); hence, the user must enter the `exit` command to terminate the shell.

`-o vi` establishes command line editing is `vi`-like. At the command prompt the user is started in *input mode* (this is different from file editing, where the user has to initiate input mode). The ESC key switches editing to *control mode*, which allows `vi`-style insert/delete and cursor movement up and down the prior command history. "Enter" sends the line currently being edited for execution and terminates the edit. <Crtl-C> aborts the edit. Command line editing is also established by assigning an editor name to the `EDITOR` or `VISUAL` keyword variable (usually limited to one of `vi`, `emacs`, or `gmacs`). Many of `vi`'s specialized control characters can be used in control mode for command line editing. For further information, consult the `man` page for `ksh`.

e) There are special shell variables that can be used in shell scripts to access useful information; e.g.,

- `$0` gives the name of the script
- `$1`, `$2`, …, `$9` reference command line arguments supplied when the script was called
- `$#` is the number of command line arguments supplied in the call
- `$*` is the list of all command line arguments supplied in the call (which can be more than 9)
- `$@` is the same as `$*` unless in enclosed in double quotes, in which case each entry in the list is treated as if enclosed in double quotes (a sometimes useful distinction, as in case of `$myvar` vs. `"$myvar"`)
- `$$` is the process id for the shell running the script (so `echo $$` identifies the process id for a shell)
- `$?` is the exit status of the last command run; hence if `exit 23` is executed to terminate a spawned shell, `echo $?` in the parent will return 23, providing a means of viewing the *exit code* returned to the calling program.

f) User versions of commands can be set up for a shell by defining an `alias` or a function of the same name that effectively reconfigures the command. Aliases can even replace commands built-in to the shell (whereas functions cannot, at least not directly). `cd` is an example of a command that is built-in to the Korn shell. To have functions and aliases apply within each spawned shell, their definitions simply need to be in the script pointed to by the environment variable `ENV`.

For example, the alias

```
alias ls="ls -Fa"
```

has the effect of changing the ls command to always apply the "F" and "a" parameters to the list request (see the man page for ls to review what these parameter are for).

If the function definition

```
man ()
  {
/usr/bin/man $* | pg -sp "page %d: "
  }
```

is in your ENV script, then whenever you execute man from a shell, this function runs instead, *piping* the man page into the pg program for display, user prompt as indicated. Since pg can look backward in a file, the alias may provide more flexible access to man pages than the original command does.

The less utility is provided on most systems and is generally considered to be an improvement to the more command and the pg command for exhibiting text file contents. Among other capabilities less allows backward movement and provides vi-like search capabilities.

LESS is a configuration variable for less. A suggested .profile entry for the LESS is

```
LESS="NCeMPM?f%f\: in page %d of %D
    (line %l of %L):piped input - in
    page %d (line %l)."
export LESS
```

(enter the given value for LESS all on one line).

g) A shell script general enough to be useful in multiple contexts is a good candidate for a function defined in `.bashrc` so that it is available across directories and spawned shells. For example,

```
list ()
{
  if [ $# -eq 0 ]; then
     ls –laF | egrep –v '^d|^l'
  elif [ "$1" = "-l" ]; then
     ls -laF | egrep '^l'
  elif [ "$1" = "-d" ]; then
     ls -laFA | egrep '^d'
  elif [ "$1" = "-dl" ] || [ "$1" = "-ld" ]
     then
     ls -laFA | egrep '^d|^l'
  elif [ "$1" = "-e" ]; then
     ls –l | egrep '\*$'
  else
/bin/ls -Fa | egrep '(^['$1'])|(^\.['$1'])'
  fi
}
```

`list` extracts `ls` information for everything but directories and links.
`list -d` and `list -l` extract `ls` information for directories and links, respectively, and
`list -ld` or `list -dl` produce `ls` information for both.
`list -e` lists names of files marked executable.
`list <`*string*`>` lists files whose names start with a character in *string* or start with a `.` followed by a character in *string*.

The `a` parameter for `ls` includes files beginning with `.` in the listing. The `A` parameter suppresses the `.` and `..` directory listings. The F parameter marks executables with a trailing *.

The `grep` filter is discussed in the context of shell programming. `-eq`, `-lt`, etc are used for number comparisons; `=` for strings.

h) Using aliases, functions, and the script language, it is possible to dynamically change the command prompt to exhibit the both the level of a spawned shell and the current working directory. Here's how it can be done:

Suppose that the ENV script includes the entries

```
export kshdepth
if [[ $kshdepth = "" ]]
  then kshdepth=0
  PS1="$tPWD $ "
else
  kshdepth=$(expr $kshdepth + 1)
  PS1="[$kshdepth] $tPWD $ "
fi
```

then whenever a new shell is spawned with ksh the command prompt will include a level number (none for the login shell, 1 for shells spawned from the login shell, 2 for shells spawned from level 1 shells, and so on). [Note: *if you install this, be sure that you don't introduce or remove spaces unless you understand the script language*!].

Note that this sets the prompt at shell start-up. It also should be changed whenever the working directory gets changed by the cd command. This means that cd needs to be redefined. Since cd is a built-in command, both an alias and a function will be needed.

Here's a solution:

```
alias cd=cdir
cdir ()
  {
    \cd $* # call is to built-in cd
    tPWD=$(echo $PWD | sed "1
                    s/home\/$USER/~/")
    if [[ $kshdepth = 0 ]]
      then PS1="$tPWD $ "
    else
      PS1="[$kshdepth] $tPWD $ "
    fi
  }
```

Note that the means for preventing a recursive call to `cdir` as the `alias` for `cd` is to use the reference `\cd`. The backslash character prevents substitution, so the built-in `cd` command is invoked rather than its `alias` (which would otherwise attempt a pointless recursion). The streaming editor `sed` is used to replace the home directory designator, if present, with "~". Command substitution (discussed earlier) is employed to set the new working directory string. For information on `sed` consult the `man` page, but it is easy to see that search and replace on line 1 (the only line supplied to `sed`) is being conducted via the line editing "`s`" command, replacing the home directory string, if found, with the string "~". (*Remark: if this is copied, the wrap-around lines in* `cdir` *caused by the narrow right page margin should be entered on a single line.*)

11. Recommended reference

Unix System V: A Practical Guide by Sobell (Benjamin-Cummins)