CS 114 - Fall 2004

Lecture 4 - Monday, October 4, 2004

Shells: Comparative Religion 102

When you login, Unix presents you with a command line that allows you to execute programs.

The core of Unix is actually not much more than a library of code. Programs make calls to functions in Unix to display information on the terminal, to write to a file, to read from a file, to interact with the network, etc.

In fact, the interaction with Unix that you perform on the command line is done with a special program that reads input from the keyboard, parses it, and invokes appropriate commands with appropriate arguments. This program is called the *command-line interpreter*, the *command shell*, or simply just the *shell*.

There are many flavors of shells, each with a different feel and with different features. Most shells are in one of two families. The *Bourne shell* family includes the following shells:

- `sh`, the original Bourne shell
- `ksh`, the `sh`-compatible Korn shell
- `bash`, the Bourne-again shell, which has features of `ksh` and also `csh` (below)

The *C shell* family includes the following shells:

- `csh`, the original C shell
- `tcsh`, an extended `csh` with more functionality

Typically, when we say something works for the Bourne shell, it will work for all shells in the Bourne shell family, and similarly for the C shell.

When you login, the system starts up an initial shell, called the *login shell*. You can change which shell is your login shell by invoking `chsh`. On `turing`, it will look something like this:

```
% chsh
** NOTE: Substituting equivalent Solaris command:
** /usr/bin/passwd -r nis -e
Enter existing login password:
Old shell: /usr/local/bin/tcsh
New shell: /usr/bin/bash
passwd: password information changed for njn2
```

The next time you login (after a few minutes delay on `turing` to update a database), your login shell will be the new shell. You can find a list of permitted login shells in the file `/etc/shells`.

Since a shell is just a program, you can always switch to a new shell by invoking the shell at the command line. For example, if your login shell is `csh`, you can start `bash` by simply executing:

```
% bash
$
```

Note that the prompt changed. Traditionally, shells in the Bourne shell family have a `$` prompt; shells in the C shell family have a `%` prompt. We can then issue commands interpreted in this new shell. We will see later in this lecture and in upcoming lectures some of the differences between the shells: how environment variables are set, I/O redirection, job control, etc.

You can exit a shell, by typing:

```
$ exit
%
```

Depending on your configuration, you can also type `^D` (that is, `control+D`) at the prompt.

The shell is what manages jobs and I/O redirection. We'll talk about these in the next lecture. The shell is also in charge of parsing the command line and performing substitutions.

Shell substitutions

When you enter a line at the shell prompt, before the shell executes programs that you specify, it will make a pass over the line you entered and perform various substitutions. One form of substitution is called *wildcard expansion* or *globbing*.

The characters `*`, `?`, and `[...]` are *wildcards*. Before a command is invoked, the shell replaces these characters with the appropriate portions of filenames. For example:

- `*` by itself is replaced by the names of files in the current directory. Thus, if you want to see the contents of all (non-hidden) files in the directory, you can run the command:

  ```
  % cat *
  ```

- `*.txt` will match all files in the current directory ending with `.txt`, for instance:

  ```
  this.txt
  and.this.one.too.txt
  ```

  `*`, in general matches zero or more characters in a file or directory name (but not the separator `/`).

- `?.txt` will match all files in the current directory whose name is a single character followed by `.txt`, for instance:

  ```
  a.txt
  2.txt
  ```

  but not:

  ```
  foo.txt
  ```

- `[abc].txt` will match the following files in the current directory:

  ```
  a.txt
  b.txt
  c.txt
  ```

  Ranges can be specified as well, for instance `[a-f]` is equivalent to `[abcdef].txt`.

- `~/[a-z]*/?` will match all files with one-character names in subdirectories of the home directory whose name starts with a lowercase letter.

Wildcards will not will not match hidden files (those beginning with `.`). To match all hidden files use "`.*`".

If you want to pass one of the wildcard characters to the command without it getting expanded, you need to *escape* it. The easiest way to escape a character is to put a backslash (`\`) in front of it. For example, the command `echo` echoes (prints) its arguments:

```
% echo *
file1 file2 file3
% echo \*
*
```

`echo` is a also good way to test if your substitutions are being performed as you intend.

Shell configuration

Every shell is setup to read from configuration files when it is started up. These files are typically used to:

- Set *environment variables* used by various programs.
- Set various shell-specific options.
- Define command *aliases*.

Configuration files

The different shells use different configuration files.

- `sh` and `ksh`: runs `.profile`
- `bash`: if a login shell, runs `.bash_profile` (or `.profile`); or if an interactive non-login shell (for example when `bash` is invoked from the command line), runs `.bashrc`
- `csh`: runs `.cshrc`, then if a login shell, runs `.login`.
- `tcsh`: runs `.tcshrc` (or `.cshrc`, then if a login shell, runs `.login`.

Note: On `turing`, you should not edit the above files. Instead, copy and configure the files in `/usr/share/skel` as specified in `/usr/share/skel/CUSTOMIZATION`. For example, if your shell is `csh` or `tcsh`, first, copy the files:

```
% cp /usr/share/skel/cshrc:SunOS.5 ~/.cshrc:SunOS.5
% cp /usr/share/skel/login:SunOS.5 ~/.login:SunOS.5
```

Then, edit `~/.cshrc:SunOS.5` and `~/.login:SunOS.5`.

For more information, do:

```
% less /usr/share/skel/CUSTOMIZATION
```

Each configuration file contains commands to initialize the shell. Since `.cshrc` may be run for non-interactive shells (such as those started to run a script), the file should not contain any interactive commands. These, should go in the `.login` file.

At startup, `bash` invokes *either* `.bash_profile` or `.bashrc`, but not both. To avoid duplicating commands in both scripts, you can set up one script to invoke the other. Put the following in your `~/.bash_profile` to run `~/.bashrc` if it exists.

```
if [ -f ~/.bashrc ]; then
    . ~/.bashrc
fi
```

If you make a change to one of these files, it will not take effect until you next login. Or, you can "source" the file as in the following examples. In the `bash` shell:

```
$ . ~/.bashrc
```

or in C shell:

```
% source ~/.cshrc
```

Sourcing a file runs the commands in the file in the current shell.

Environment variables

Many programs use *environment variables* to control their behavior. For example, if you look at the man page for the program `less`, you'll discover it uses the environment variable `LESS` to communicate command line options to pass automatically to `less`. The command line option `-F`, for instance, tells `less` to exit if the entire output fits on one screen. To pass this argument to `less` each time it is invoked, you can set `LESS` to `-F`. In the Bourne shell `sh`, this is done with:

```
$ LESS=-F
$ export LESS
```

In `bash` and `ksh`, these commands can be shortened to just:

```
$ export LESS=-F
```

The following is the equivalent in the C shell:

```
% setenv LESS -F
```

To set this environment variable each time you login, you can set the variable in your `.bashrc` or your `.cshrc` simply by adding the appropriate command above to the file.

You can learn the value on an environment variable by running, for example:

```
% echo $LESS
-F
```

The `env` command lists all defined enviroment variables.

Other useful environment variables include:

- `PAGER` – set to the program to use as the default pager. If not set, `more` is used. You might want to set `PAGER` to `less`:

  ```
  $ export PAGER=less
  ```

  or:

  ```
  % setenv PAGER less
  ```
- `EDITOR` and `VISUAL` – these environment variables define the default editor to use for editing files. Set to `vi`, `vim`, or `emacs` to suit your preference.
- `HOME` – this is the path to your home directory; it should be set automatically by the shell
- `LOGNAME` – this is your login name; it should be set automatically by the shell
- `SHELL` – this is path to the shell you are running; it should be set automatically by the shell
- `HOST` – your hostname; it should be set automatically by the shell
- `PATH` – this variable specifies the list of directories to search when invoking a command. This variable deserves it's own section.

PATH

When you invoke a command at the command line, you can either specify its full path, e.g.,

```
% /bin/ls
file1 file2 file3
```

Or, you can let the shell search for the command:

```
% ls
file1 file2 file3
```

The shell searches the directories in the environment variable `PATH` in order for the first executable file with the given name. The `PATH` contains a list of directories separated by colons (`:`). For example, a typical `PATH` might be:

```
% echo $PATH
/usr/local/bin:/bin:/usr/bin:/usr/bin/X11:/usr/local/gnu/bin
```

In the Bourne shell you set `PATH` by doing:

```
$ PATH=/usr/local/bin:/bin:/usr/bin:/usr/bin/X11:/usr/local/gnu/bin
```

Since `PATH` is somewhat important, in the C shell you can set the `PATH` with `setenv`, or with the special syntax:

```
% set path = ( /usr/local/bin /bin /usr/bin /usr/bin/X11
/usr/local/gnu/bin )
```

If you need to temporarily add a directory to your `PATH`, you can append to it as follows:

```
$ PATH=$PATH:/usr/ucb
```

or:

```
% setenv PATH $PATH:/usr/ucb
```

Aliases

Another thing to put in your shell configuration files are *aliases*. An alias is simply a anme that you can use as a command, but *expands* to a more complicated command before being executed. For example, since I frequently use the command `ls -l`, I have aliased it to `ll`. The original Bourne shell and the original C shell do not support aliases, but `bash`, `ksh`, and `tcsh` do. The alias related commands in the Bourne shell family are:

- `alias` lists all the defined aliases
- `alias` *`name`* lists the alias, if any, for *`name`*
- `alias` *`name=value`* sets an alias.
  You may need to enclose *`value`* in single- or double-quotes if it includes a space.
- `unalias` *`name`* deletes an alias
- `unalias -a` deletes all aliases

The C shell family is similar:

- `alias` lists all the defined aliases
- `alias` *`name`* lists the alias, if any, for *`name`*
- `alias` *`name value`* sets an alias
- `unalias` *`name`* deletes an alias

Some aliases you might find useful include:

- `ll` for `ls -l`
- `la` for `ls -a`

`ls` `ls -sCF`
- `rm` for `rm -i`
- `l` for `less`
- `rd` for `rmdir`
- `md` for `mkdir`
- `c` for `clear`
- `dc` for `cd` (a common typo)
- `mroe` for `more` (another common typo)
- `bye` for `exit`

Prompts

You can change the command prompt in the Bourne shell family by setting the `PS1` environment variable.

You can change the command prompt in the C shell family by running, for example:

```
% set prompt = '%n@%m %~ %% '
njn2@turing ~ %
```

Here, `%n` is replaced with your username; `%m` is replaced with the hostname; `%~` is replaced with the current directory; `%%` is just a literal "`%`". Read the man pages for your shell for more details.

umask

Suppose you want to prevent other users from accessing your files. If you're conscientious, you can check the permissions often and `chmod` to correct them. But, you can also set the default permissions for newly created files by using the `umask` command in your shell startup file.

`umask` takes a *permission mask* as an argument. Recall that `chmod` can take numeric permission arguments like 755 (rwx for the user, rx for the group and others), or 640 (rw for the user, r for the group, no permission for others). When a file is created the umask is used to *remove* permissions from the new file. For example:

- `umask 022` prevents the group and others from writing your files
- `umask 066` prevents the group and others from reading or writing your files
- `umask 077` prevents the group and others from reading, writing, or executing your files
- `umask 027` prevents the group writing your files, and all others from reading, writing, or executing your files