C Programming under Linux

P2T Course, Semester 1, 2004–5
Linux Lecture 2

Dr Graeme A Stewart

Room 615, Department of Physics and Astronomy

University of Glasgow

`graeme@physics.gla.ac.uk`

Summary

- Interactive Shell Features
- Unix Help Systems
- Shell Aliases
- Unix Shells
- Shell Commands
- PATH and Other Variables
- Shell Startup and Shutdown

`http://www.physics.gla.ac.uk/p2t/`

`$Id: linux-lecture-02.tex,v 1.17 2004/10/15 09:38:11 graeme Exp $`

Command History

Before we go any further, it's worth knowing that `bash` keeps a history of typed commands.

- Press the up-arrow ↑ to get the last command
- Press the down-arrow ↓ to get the previous command
- All of the recovered commands are editable:
  - `CTRL-A` Beginning of line
  - `CTRL-E` End of line
  - `ALT-F` Forward one word
  - `ALT-B` Backwards one word
  - `ALT-D` Delete one word

These commands are the same as those found in the famous Emacs text editor, originally written by Richard Stallman.

Tab Completion

`bash` has a great feature for command line use, called *tab completion* (first found in `TENIX`, later in `tcsh`).

- If you start typing a command name and press TAB then. . .
  - if the typed characters are unique to a command `bash` will fill in the rest of the name.
  - if the typed characters are not unique `bash` will sound the bell – keep typing or press TAB again for a list of possible completions.
- Continue on and give filename arguments to a command and the same applies.

```
$ con[TAB]                                     #  Shell bleeps
     [TAB]                                     #  TAB again - possible completions
configurewrapper  consolechars      convcal
conjure           continue          convert
$ conve[TAB]                                   # Type "ve"
  convert                                      # Shell completes command
```

Long Lines, Short Lines

The shell will quite happily read very long lines which wrap past a single terminal line, but sometimes this is not convenient.

Long lines can be split if the last character is a `\`:

```
$ echo this line\
> is split\
> over quite a few\
> input lines
this line is split over quite a few input lines
$
```

Notice that the shell changes the prompt from $ to > when it's waiting for a long command to be finished.

If you separate them with a `;` you can type more than one command on a line:

```
$ echo here we go; echo here we go again...
here we go
here we go again...
```

Comments

And the # character is a comment character for the shell: it, and anything which comes after, is ignored:

```
$ # this is completely ignored
$ echo and this line # is partly a comment
and this line
```

If you actually want a # printed, precede it with a backslash or enclose it in quotes:

```
$ echo this will print a \# character
this will print a # character
$ echo "and here is another way to print a # character"
and here is another way to print a # character
```

These are examples of *escaping* and *quoting* – more on them later.

Wildcards

As we saw, lots of commands can take multiple arguments. The shell can help you select groups of files by using *wildcards*:

- `*` Matches any number of characters (even none)
- `?` Matches any single character (but only one)

Here are some wildcard examples...

```
$ ls
foo.jpg    bar.jpg     baz.jpg    nip.jpg
foo.tex    foo.tex~    bar.tex    baz.tex    baz.tex~
$ ls *.jpg
foo.jpg    bar.jpg     baz.jpg    nip.jpg
$ ls *~
foo.tex~   baz.tex~
$ ls ?a?.jpg
bar.jpg    baz.jpg
$ rm *
$ ls
$
```

Wildcards II

Notice that `*` on its own matches *everything* in a directory (except things beginning with "`.`").

If you want the shell to *ignore* a wildcard, preceed it with a `\` or enclose it inside quotes: `""` or `' '`

```
$ ls
foo**bar        foobazbar        foo?bar       fooAbar
$ ls foo*bar
foo**bar        foobazbar        foo?bar       fooAbar
$ ls foo?bar
foo?bar         fooAbar
$ ls "foo?bar"
foo?bar
```

This is important if you want the command you are executing to see the wildcard.

Note that `*` and `?` are legal filename characters in unix – in fact the only forbidden characters are the directory separator `/`, and the `NULL` character (`\000` or `0x00`). It's certainly not recommended to create such filenames, but they can be handled!

Getting Help

No one could accuse unix commands of being timid or intuitive (at least until you start to dream in bytecode. . .). In general they are *powerful* and *terse* to the point of obscurity; designed for efficiency rather than comfort.

They also can take a bewildering array of options (e.g., about 65 to `ls`), fine tuning their output of behaviour to exactly that required.

No one is really expected to *remember* the behaviour of unix commands in minute detail (after all, that's the computer's job!), so Linux has a number of *help systems* to aid you in

- finding the commands that you need
- documenting how to use them

Help I: man

Basic help on unix commands is provided through the ‘manual pages’ system. This is invoked by the command `man`:

```
 $ man ls
 LS(1)              User Commands             LS(1)
 NAME
       ls - list directory contents
 SYNOPSIS
       ls [OPTION]... [FILE]...
 DESCRIPTION
       List  information about the FILEs (the cur-
       rent directory by default).   Sort  entries
       alphabetically  if  none  of  -cftuSUX  nor
       --sort.
       Mandatory arguments  to  long  options  are
       mandatory for short options too.
       -a, --all
              do  not hide entries starting with .
[...]
       -l     use a long listing format
```

Help II: man

Having discovered the `-l` option to `ls`...

```
$ ls Makefile
Makefile
$ ls -l Makefile
-rw-r--r--  1 graeme knigits  514 2003-09-11 21:56 Makefile
```

You'll see that man pages are information dense (they are designed for efficiency rather than comfort, just like the commands they describe). But they all have the same structure: name of command, synopsis, description, details.

Also they have a really useful section at the end called `SEE ALSO`, which lists related commands. From `man rm`

```
SEE ALSO
       shred(1)
```

Man pages automatically use the pager `less`, so all the navigation and searching options of `less` work.

Help III: man

If you are looking for a command to do something, but you can't quite remember its name, then you can use the `apropos` command (equivalent to `man -k`):

```
$ apropos remove
rm (1)                - remove files or directories
rmdir (1)             - remove empty directories
uniq (1)              - remove duplicate lines from a sorted file
remove (3)            - delete a name and possibly the file it refers to
```

The number after each entry (and at the very top of the `man` listing) refers to the section of the manual. See `man man` for the complete list, but we will be most interested in commands in section 1, 'Executable programs or shell commands'.

The man pages in section 3, 'Library calls (functions within system libraries)' will be useful for the `C` part of this course.

Help IV: Shell Builtins

A lot of commands we work with are *shell builtins*, not external commands (e.g. `cd`, pwd, `kill`, etc.).

To get help with these commands either read the shell manpage: `man bash`, or use the builtin `help`:

```
$ help for
for: for NAME [in WORDS ... ;] do COMMANDS; done
   The ‘for’ loop executes a sequence of commands for each member in a
   list of items.  If ‘in WORDS ...;’ is not present, then ‘in "$@"’ is
   assumed.  For each element in WORDS, NAME is set to that element, and
   the COMMANDS are executed.
```

`help` on its own lists all the builtins that `help` can provide information about.

Shell Aliases

Man pages and other forms of help are great for finding out what command options are avaliable, but it's pretty tedious to type `ls --color=auto` each time you want the shell to colorise the output from `ls`.

Fortunately the shell allows the definition of *aliases*, which can act a shortcuts for larger definitions of commands and options.

```
$ alias lc="ls --color=auto"
$ lc
      a.out       README       teaching       work
```

In fact we can alias commands to themselves, which is better if we decide that colorised `ls` is just a grand idea:

```
$ alias a=alias
$ a ls="ls --colour=auto"
$ a v="ls -l"
$ v a.out
-rwxr-xr-x 1 graeme knigits 20034 2003-10-01 21:22 a.out
```

The Basic Shell

We’ve seen that the basic user interface in unix is *the shell*.

However, although the shell behaves in a simple and expected way, underlying it there is a much more sophisticated interface with far more powerful capabilities.

A History of Unix Shells

- The first significant unix shell, written by Stephen Bourne, was `sh` (or `/bin/sh`). It was released in 1979 with AT&T's V7 UNIX. It was great for programing, but poor for user interaction. Neverthless it became the basic unix shell, and all compatible shells are called *Bourne Shells*.
- The next major dialect of shell came with the release of Berkley 2BSD UNIX. This shell, `csh` or `/bin/csh`, and its compatible descendents are known as *C Shells*.
- The C shell became particularly popular in academia – it was an easier shell to use than the original `sh`; however, it had a number of drawbacks for programing system scripts, where `sh` remined dominant.
- A third shell dialect, the `Korn Shell` (released with SRV 4 unix in 1988), also exists. It's really a superset of the Bourne shell. It is less popular than the `sh` or `csh` as its code is still proprietary.

Shells on a Linux System

All linux systems will contain, at minimum, a version of `sh`: `sh`, `ash`, `zsh`, `bash`; and usually a basic `csh` too.

We will use `bash` in this course, the GNU *Bourne Again SHell*.

- Bash is a Bourne Shell (as you might expect).
- However, bash also offers the ease of use of modern C shells.
- `bash` has become the *de facto* standard shell in all popular Linux Distributions.
- The path to bash is `/bin/bash`.

There are a few extensions which `bash` provides over the standard provisions of '`sh` classic', but it is compliant with the new `POSIX 1003.2` shell standard.

We'll try and indicate where important differences with other `sh` shells lie, and what remedial action one might take.

Binaries, Paths and Builtins I

We have seen how the shell runs commands on behalf of a user. Most of these commands are *binaries* which exist in the linux distribution. e.g. the command `ls` is a binary which lives in `/bin`:

```
$ ls -l /bin/ls
-rwxr-xr-x  1 root   root   69228 2003-08-19 01:50 /bin/ls
```

When a user types `ls` how does the shell know where to find the command?

The shell has an important variable, called PATH which is a colon separated list of directories on the system where the shell will search for commands:

```
$ echo $PATH
/bin:/usr/bin:/usr/X11R6/bin:/usr/local/bin
```

So, when a user types `ls` the shell first searches for `/bin/ls`, then `/usr/bin/ls`, then `/usr/X11R6/bin/ls` and so on.

The first match wins! (In this case `/bin/ls`.)

Binaries, Paths and Builtins II

The command `type` will print the shell's path to a command:

```
$ type less
less is /usr/bin/less
$ type gv
gv is /usr/X11R6/bin/gv
```

The shell will, for speed, cache the location of commands which it has already found. We can get the shell to forget its current cache using the command `hash -r`. (This can be important if the `PATH` is changed.)

The shell also interprets some commands itself. These commands are called *shell builtins*:

```
$ type cd
cd is a shell builtin
```

There's usually no man page for shell builtins – instead they are documented in the shell manpage (`man bash`) and via the shell's help system: (`help cd`).

Shell Variables I: PATH

We've seen how the shell searches for commands, by referencing the PATH variable. Naturally, as it's a *variable*, we can change its value:

```
$ PATH=/home/graeme/bin:/usr/local/bin:/usr/bin:/bin
$ echo $PATH
/home/graeme/bin:/usr/local/bin:/usr/bin:/bin
```

This will change how the system looks for commands.

Notice the difference between setting a variable, where we use NAME=VALUE, (*N.B.* there can't be any spaces around the = sign) and referencing its value, $NAME:

```
$ echo PATH
PATH
$ echo $PATH
/home/graeme/bin:/usr/local/bin:/usr/bin:/bin
```

Shell Variables II

Although the PATH variable's meaning is special, it's just another variable in the shell, and we can set any variable in the same way:

```
$ MSG=the-wild-deer-eat-cheese
$ echo $MSG
the-wild-deer-eat-cheese
$ SHOUT=echo
$ $SHOUT $MSG
the-wild-deer-eat-cheese
```

As we've seen, to print a variable's value use `echo`. To print all shell variables use the builtin `set`:

```
$ set
BASH=/bin/bash
COLUMNS=80
CVSROOT=:ext:login.astro.gla.ac.uk:/home/graeme/data/cvs
CVS_RSH=ssh
DIRSTACK=()
DISPLAY=:0
EUID=1000
[...]
```

Shell Variables III: Ambiguity

If there’s some ambiguity about a variable’s name, use curly braces:

```
$ MSG="inside of a dog it’s too dark to "
$ echo $MSGread

$ echo ${MSG}read
inside of a dog it’s too dark to read
```

Notice the use of quotes when the value of the variable contains spaces. Also notice, importantly, that an undefined variable expands to an empty string (and this is not an error!).

Legal characters in shell variables are alpha numerics, plus the underscore (_), but the first character cannot be a number:

```
$ good_2_go=yes
$ 2_good_2_go=no
bash: 2_good_2_go=no: command not found
```

More About Quoting

We saw previously that we could *escape*, using `\`, or *quote* to avoid the shell misinterpreting special characters like $ or `*`.

However, there are two types of quotes `"double"` and `'single'`. The difference is that:

- Inside double quotes variables are expanded
- Inside single quotes nothing is altered or expanded at all

```
alpha$ a=spam; b="$a"; c='$a'
alpha$ echo "$a $b $c"
spam spam $a
alpha$ echo '$a $b $c'
$a $b $c
```

Note that inside double quotes single quotes are just characters, and *vice versa*, so single quotes are particularly useful for commands which themselves use quotes, like `perl`:

```
$ perl -ne '@e=split(/:/, $_); print "$e[2]\n" if $e[4] =~ /^Person/;' \
  < contacts
```

Exporting Variables

The shell treats all variables pretty much the same, however, there is a ‘flag’ on some variables, which tells the shell to pass the definition of that variable on to commands and subshells. This then distinguishes between ‘local’ and ‘global’ variables.

To set a variable for export use `export`:

```
$ VAR=SOMEVALUE
$ echo $VAR
SOMEVALUE
$ bash -c ’echo $VAR’

$ export VAR
$ bash -c ’echo $VAR’
SOMEVALUE
```

If in doubt, you probably want to export a variable!

`export` (no arguments) will also list of all variables marked for export:

```
$ export
declare -x CVSROOT=":ext:login.astro.gla.ac.uk:/home/graeme/data/cvs"
declare -x CVS_RSH="ssh"
declare -x DISPLAY=":0"
[...]
```

Shell Startup Scripts

When a `bash` shell begins it executes various commands which it finds in files held in the user's home directory.

Technically this is called *sourcing*, and it means that the contents of the file are read and processed as if they were being typed in on the command line.

- If the shell is a *login shell* it sources `~/.bash_profile`
- If the shell is not a login shell it sources `~/.bashrc`

These files are excellent places to customise your shell environment.

.bash_profile and .bashrc

`.bash_profile` and `.bashrc` contain commands which customise the user's shell environment – setting `PATH`, environment variables, shell aliases, etc.

```
$ head -6 .bash_profile
# .bash_profile for login shells
PATH=~/bin:$PATH:/usr/local/bin
LC_ALL=en_GB
CVSROOT=:ext:login.astro.gla.ac.uk:~/data/cvs
CVS_RSH=/usr/bin/ssh
export CVSROOT CVS_RSH
```

`.bashrc` tends to contain aliases:

```
$ head -6 .bashrc
alias a='alias'
alias e='emacs'
alias h='history | tail -20'
alias hs='history | grep $*'
alias j='jobs'
alias ls='ls --color=auto'
alias m='less'
```

.bash_logout

When a *login* shell exits, it will source `.bash_logout`. Usually this isn’t terribly important, but it can be used to kill helper programs, e.g. `ssh-agent`:

```
$ cat .bash_logout
if [ -n "$SSH_AGENT_PID" ]; then
       echo Killing ssh-agent
       kill $SSH_AGENT_PID
fi
echo "Farewell and Godspeed $logname!"
```

Sourcing in General

We discussed how a file, like `.bash_profile` is *sourced*, i.e., interpreted as a series of commands, as if they were typed at the shell prompt.

This is a useful thing to be able to do at other times, e.g., some software will require a complex set of environment variables. In this case use the builtin `source`:

```
$ cat sourceme
export SSW=/usr/local/ssw
$ source sourceme
$ echo $SSW
/usr/local/ssw
```

There is an ‘alias’ for source: a single dot:

```
$ . sourceme
$ echo $SSW
/usr/local/ssw
```

Copyright