C Programming under Linux

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

Dr Graeme A Stewart

Room 615, Department of Physics and Astronomy

University of Glasgow

`graeme@physics.gla.ac.uk`

Summary

- Text Manipulation Commands
- File Type
- Searching for Files
- Command Substitution
- Shell Odds and Ends
- Unix Permissions

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

`$Id: linux-lecture-05.tex,v 1.6 2004/10/26 11:34:46 graeme Exp $`

Text Manipulation: Heads or Tails?

Let's look now at some more command line utilities for manipulating text files – some of these you'll have met now in the labs.

To look at just beginning or the end of a text file, use the commands `head` or `tail`.

- `head [-N]` print the first `N` (default 10) lines of input
- `tail [-N]` print the last `N` (default 10) lines of input

`tail` has another very useful option, `-f`, which keeps on trying to read from a file (until interrupted):

```
$ frobnicate > output.txt &
[1] 23111
$ tail -f output.txt
frobnicate started Sat Oct  4 08:22:25 BST 2003.
processing star fangled mingleweep list.
[...]
```

Text Manipulation: Sorting

To sort text files we use the splendidily obvious command `sort`.

```
$ echo -e "sort should\nalphabetise these\nlines. not\n"\
> "really so useful\nin this case" | sort
alphabetise these
in this case
lines. not
really so useful
sort should
```

The most useful options to sort are:

- `-n` Sort numerically instead of alphabetically
- `-k N` Sort the `N`th ‘key’
- `-t CHAR` Separate fields with character `CHAR` instead of whitespace.
- `-r` Reverse sort (largest first)

Text Manipulation: uniq

In parallel with `sort`, we can use `uniq`. This takes a sorted list and removes any duplicate lines:

```
$ cat list
a
bb
bb
c
$ uniq list
a
bb
c
```

A more useful example would be removing duplicate error messages from log files:

```
$ grep "^Error:" *.log | sort | uniq
```

Text Manipulation: Greping

Another very useful unix command is `grep`, *Get REgular exPression*. This prints out lines from its input which match its criterion

```
$ grep ssh /etc/services
ssh             22/tcp                          # SSH Remote Login Protocol
ssh             22/udp                          # SSH Remote Login Protocol
```

`grep`, like most unix commands, has a multiplicity of options (as ever, see `man grep`). Some of the most notable are:

- `-v` Only print lines which do not match
- `-n` Print line numbers of matching lines
- `-N` Print N lines before and N lines after each match (i.e. give context)
- `-i` Ignore case (i.e. UPPER and upper are the same)

Text Manipulation: Regular Expressions I

`grep`'s real power comes from the ability to match *regular expressions*, which are a little like the shell's wildcards, but are much more powerful.

`grep` has two regular expression modes: we'll use the *extended* mode, invoked by `grep -E` or `egrep` (the syntax is more useful than 'original' `grep`, which is now considered anachronistic).

- Regular expressions are built up of single character matches: most characters just match themselves, so `grep spam` matches character `s`, then `p`, then `a` then `m`.
- A dot, `.`, matches any character at all.
- Any group of characters inside `[]`s match any one of those characters: so `egrep '[rbz]ed'` will match `red`, `bed` or `zed`. N.B. `rbed` is *not* matched.
- If the `[]`ed expression starts with `^` then it matches any characters *not* listed: `egrep '[^rbz]ed'` will match `aed`, `Ced`, `4ed`, `=ed`, etc.

Text Manipulation: Regular Expressions II

- Inside a `[]`ed expression characters separated by a hyphen match those characters and any in between, in the character set used: `[a-z]` matches any lower case letter; `[2-6]` matches 2, 3, 4, 5 or 6; `[A-E0-9]` matches any hex character.

A single match can then be modified in several ways by a repetition operator:

- `?` Optional match – don't match or match once.
- `*` Match zero or more times.
- `+` Match one of more times.
- `{n}` Match n times.
- `{n,}` Match n or more times.
- `{n,m}` Match between n and `m` times.

Text Manipulation: Regular Expressions III

Finally, a regular expression can be *anchored*:

- ˆ at the beginning of an expression anchors it to the beginning of a line.
- $ at the end of an expression anchors it to the end of a line.

```
$ grep -E 'ˆ[a-z]'              # Match any line beginning lower case
$ grep -E 'gnu*s?'              # Match 'gn', 'gns', 'gnus', 'gnuu', ...
$ grep -E 'ˆ[0-9A-Fa-f:]+$'     # Match any line wholely of hex characters and :s
$ grep -E 'ˆ[ˆ#]'               # Match any line not beginning with a #
$ grep -E 'ˆstart.*end$'        # Match any line beginning with 'start' and
                                #  ending with 'end'
                                #  (With anything inbetween:
                                #   .* = any character, any number of times)
```

N.B. It's a good idea to enclose a regular expression match in quotes to stop the shell expanding its wildcards.

Text Manipulation: wc

The command `wc` will *word count* files given on the command line, or `STDIN`:

```
$ wc lear.txt
  4847  27580 150870 lear.txt
```

The output from `wc` with no switches lists the number of lines, words and characters in each file.

A particularly useful option to `wc` is the `-l` switch, which prints only the number of lines:

```
$ grep -i king lear.txt | wc -l
330
```

So, `lear.txt` contains 330 lines with the word ‘king’ (case insensitive).

File Type

Unix has a very useful command, called `file`, which will attempt to find out the type of file. It prints a text string decribing the filetype.

```
$ file wireless-signup.png
wireless-signup.png: PNG image data, 693 x 722, 8-bit/color RGB, non-interlaced
$ file 5147.doc
5147.doc: Microsoft Office Document
$ file foo
foo: Bourne shell script text executable
$ file /bin/ls
/bin/ls: ELF 32-bit LSB executable, Intel 80386, version 1 (SYSV), for GNU/Linux 2.2
dynamically linked (uses shared libs), stripped
```

The text description can be grepped, e.g., in a shell script:

```
if file $image | grep "JPEG Image Data" &> /dev/null; do
  processjpeg $image
fi
```

Finding Files

Commands like `ls` list the files in one or more directories, but searching for files on the system requires other commands (each node in the student cluster presently has about 74 000 files).

The command `find` will search for files and directories that match against a range of criteria:

```
FIND(1L)                                                    FIND(1L)


NAME
       find - search for files in a directory hierarchy


 SYNOPSIS
       find [path...] [expression]
```

`[expression]` can take many, many options but the most useful are. . .

Find Options

- `-type f` a file
- `-type d` a directory
- `-name NAME` named `NAME` (wildcards supported)
- `-user USERNAME` owned by user USERNAME
- `-mtime N` modified N days old
- `-mtime +N` modified more than N days old
- `-mtime -N` modified less than N days old
- `-mmin ...` as `mtime`, but in minutes

Find Examples

To find all the `C` source files beneath the current directory:

```
$ find . -name "*.c" [-type f]
```

Note that without the `-type f` option we could match any directories ending in `.c`.

To find all the `C` files modified in the last 4 days in `/home/graeme` and `/home/ralf`:

```
$ find /home/graeme /home/ralf -name "*.c" -mtime -5 [-type f]
```

To find all `C` source files owned by user `ptang` in directory `/usr/local/src` modified less than an hour ago:

```
$ find /usr/local/src -type f -user ptang -name \*.c -mmin -60
```

Notice how multiple options are logically `and`ed together.

Finding by Name: Locate

You'll see in the lab that `find` works really slowly – it's got to look through every file and directory to search. To speed things up there's a command called `locate`, which uses a database to search for files by name:

```
$ locate stdlib.h
/usr/include/g++-v3/bits/std_cstdlib.h
/usr/include/stdlib.h
```

A simple name (without wildcards) matches any part of a file's name or path (`locate usr` will match thousands of files). With wildcards you need to match the whole name and path:

```
$ locate "*/stdlib.h"
/usr/include/stdlib.h
```

Remember that locate uses a database: that makes it fast, but it may be out of date (usually the database is built everynight - see `/etc/cron.daily/find` on our cluster. In addition *networked disks* are not usually included in the database (although on our cluster `/home` and `/usr/local` are).

Command Substitution

Sometimes we may want to do something with the output of a command other than print it to the screen. We have seen how to redirect output to a file with redirection operators, and how to redirect to another command with a pipe.

Another way to process the output from a command is to use *command substitution*. This takes the output from a command and treats it as if it were what is given on the command line or in the script. Command substitution is indicated by `$(command)`.

```
$ date
Tue Nov 10 10:09:27 GMT 2003
$ now=$(date)
$ echo $now
Tue Nov 10 10:09:29 GMT 2003
```

N.B. The `$(command)` syntax is not supported by all Bourne shells – sometimes backquotes are used: `‘command‘`.

More Command Substitution

Command subsititution is very useful when scrpting:

```
#! /bin/bash
#
# Search for all data files and frobnicate.
for datafile in $(find /data -type f -name \*.dat); do
    echo Processing file $datafile at $(date)
    frobnicate $datafile
done
```

Notice that we could not have used a wildcard, as in `/data/*.dat` – that would not have searched subdirectories.

```
#! /bin/bash
#
# Analyse unique log file errors
for error in $(grep "^Error" *.log | sort | uniq); do
    analyse $error
done
```

Conditional Execution

We saw in scripts how the shell could conditionally execute code based on the `$?` return value. However, the shell has two quick operators which are handy for this: `&&` and `||`.

`CMD1 && CMD2`	AND execution: only execute `CMD2` if `CMD1` exited successfully.
`CMD1 || CMD2`	OR execution: only execute `CMD2` if `CMD1` had a non-zero exit status.

Only `cd` if `mkdir` works...

```
$ mkdir /tmp/process && cd /tmp/process
```

Print a message if no errors are found in the logs...

```
$ grep -E "^Error" process.log || echo No errors found
```

Getting STDIN: read

A shell script can read lines from `STDIN` using the `read` command. At its most basic it reads one line from `STDIN` and assigns it to a variable:

```
#! /bin/bash
# An irritating shell script.
finish=0;
while [ $finish != "1" ]; do
    read LINE
    echo You said: $LINE
    if [ "$LINE" = "go away" ]; then
         finish=1
    fi
done
```

Remember however that `STDIN` could be another process:

```
$ frobnicate foo | error_logger
$ cat error_logger
#! /bin/bash
while true; do
  read LINE || exit 0
  echo $LINE | grep -q error && echo Error detected at $(date): $LINE >> log
done
```

Unix Permissions

How does a Linux system know who should be able to write to a file? Can I write to someone else files? How are system files protected from malicious users?

Every file in Linux has a *user* and a *group* who own it:

```
$ ls -l linux-lecture-01.tex
-rw-r--r--  1 graeme knigits  25167 2003-10-06 22:32 linux-lecture-01.tex
                ^      ^
                |       \
                 \       - group
                  - owner
```

Permission to access a file is divided into three: permissions accorded to the *owner*, the *group* and *everyone else*.

Separate permissions which can be controlled are permission to `read` the file, `write` to the file or `execute` the file.

Permissions II

`ls -l` tells us who can do what to a file:

```
$ ls -l linux-lecture-01.tex /bin/ls
-rwxr-xr-x    1 root   root     72460 2003-10-05 00:10 /bin/ls
-rw-r--r--    1 graeme knigits  25167 2003-10-06 22:32 linux-lecture-01.tex
|\^/\^/\^/
| |  |  |
|  \  \  - permissions for everyone else (or "world")
 \  \  - permissions for group
  \  - permissions for owner
   - 'file' type (- = file, d = directory, l = soft link)
```

If permission is granted on a file the relevant `rwx` bit is set and `r`, `w` or `x` is printed. If permission is not granted the permission bit is not set and `-` is printed instead.

Execute permission allows the file to be run as a command.

(There are some other bits: set uid, gid and sticky bits, but they're are more relevant to system administrators.)

Permissions III

Directories have the same properties as files: they have an owner, a group and three permission bits, `rwx`, for each catagory. However the bits have slightly modified meanings:

- `r` Means the contents of the directory can be listed
- `w` Means the contents of the directory can be modified – so files can be deleted or created.
- `x` Means that files in the directory can be accessed.

If a directory has `--x` bits set then it cannot be `ls`ted, but files which one knows the name of can be accessed.

Changing Permissions

Permissions are changed with the `chmod` command. This command takes three bits of information, in the form `catagory +/- permission`, then a list of files and directories. e.g.

```
$ ls -l foo.c
-rw-r--r--    1 ptang   mrlp     7240 2003-10-06 09:15 foo.c
$ chmod g+w foo.c
$ ls -l foo.c
-rw-rw-r--    1 ptang   mrlp     7240 2003-10-06 09:15 foo.c
```

catagory can be

u owner, or user	g group
o other, or world	a all

Then

+ means grant a permission
- means revoke it

.

Finally, a permission, or set of permissions, is listed: `r`, `w` and `x`.

You have to be the owner of a file and have write access to its directory to change its permissions.

More chmod examples

```
$ ls -l circus
 -rw-r--r--    1 monty  python    37240 2003-10-25 09:15 circus
$ chmod og-r circus; ls -l circus
 -rw-------    1 monty  python    37240 2003-10-25 09:15 circus
$ chmod u+x circus; ls -l circus
 -rwx------    1 monty  python    37240 2003-10-25 09:15 circus
$ chmod a+rx circus; ls -l circus
 -rwxr-xr-x    1 monty  python    37240 2003-10-25 09:15 circus
$ chmod g+w circus; ls -l circus
 -rwxrwxr-x    1 monty  python    37240 2003-10-25 09:15 circus
$ chmod a-rwx circus; ls -l circus
 ----------    1 monty  python    37240 2003-10-25 09:15 circus
$ cat circus
cat: circus: Permission denied
$ chmod u+r circus; ls -l circus
 -r--------    1 monty  python    37240 2003-10-25 09:15 circus
```

Permissions: Users and Groups

If you want to know which groups a user belongs to use the command `id`. By default your own user id is queried:

```
$ id
uid=1000(graeme) gid=1000(knigits) groups=1000(knigits),4(adm),5001(devel)
$ id fiona
uid=1001(fiona) gid=1001(anthrax) groups=1001(anthrax),5001(devel)
```

Then to change the group owner of a file, use *chgrp*:

```
$ ls -l foo.c
-rw-rw-r--    1 graeme  knigits   7240 2003-10-06 09:15 foo.c
$ chgrp devel foo.c
$ ls -l foo.c
-rw-rw-r--    1 graeme  devel     7240 2003-10-06 09:15 foo.c
```

The equivalent command to change file ownership is `chown`, but usage of it is a priviledge of system administrators!

Default Permissions: umask

When a file is created there is a shell setting, called the `umask` that determines which permissions it gets created with:

- `umask 022` means unwritable by group and world
- `umask 002` means unwritable by world
- `umask 027` means unwritable by group, no permissions at all for world

The `umask` determines which bits get unset (unmasked) when the file or directory is created. It's usually expressed as a 3 digit octal number, one digit for the owner, one for the group, one for world.

The value of each digit in the umask is calculated by adding a bit mask value for each permission:

$4 (= 2^2)$	$2 (= 2^1)$	$1 (= 2^0)$
`read`	`write`	`execute`

Default Permissions: Octal Mode

Remember that the `umask` is determining the permissions which are *not* granted, so:

```
$ umask 022; touch foo; ls -l foo
-rw-r--r--   1 graeme  knigits      0 2003-10-06 23:57 foo
$ umask 027; mkdir bar; ls -l bar
drwxr-x---   1 graeme  knigits   4096 2003-10-06 23:58 foo
$ umask 077; touch private; ls -l private
-rw-------   1 graeme  knigits      0 2003-10-06 23:58 private
```

Note that files are usually created without x permissions – it usually makes no sense to execute a text file. If the file is, say, a script then add execute permission manually:

```
$ emacs foo.sh
$ ls -l foo.sh
-rw-r--r--   1 graeme  knigits     526 2003-10-09 21:58 foo.sh
$ chmod a+x foo.sh; ls -l foo.sh
-rwxr-xr-x   1 graeme  knigits     526 2003-10-09 21:58 foo.sh
```

Files which are created and *should* be executable, generally are:

```
$ gcc -o foo foo.c; ls -l foo
-rwxr-xr-x   1 graeme  knigits   34526 2003-10-09 09:58 foo
```

Copyright