# C Programming under Linux

## *P2T Course, Semester 1, 2004–5*
## *Linux Lecture 3*

Dr Graeme A Stewart

Room 615, Department of Physics and Astronomy

University of Glasgow

`graeme@physics.gla.ac.uk`

## Summary

- File Descriptors
- Job Control
- Unix System Information
- XEmacs Basics

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

**`$Id: linux-lecture-03.tex,v 1.7 2004/10/15 09:38:11 graeme Exp $`**

# Out with the Shell

We've seen that commands generally send their output to the screen. In fact the command is actually printing to a *file descriptor* called *Standard Out*, or `STDOUT`.

By default this goes to the screen, but we can send it to a file by using a *redirection operator*:

```
$ ls
bar   foo
$ ls > listing
$ cat listing
bar
foo
$ ls
bar   foo   listing
```

If file `listing` existed we would overwrite its contents with > (this is called *clobbering*).

# Append and Input Redirectors

To avoid clobbering we can to append to a file using *append* operator, >>:

```
$ ls >> listing
$ cat listing
bar
foo
bar
foo
listing
```

We can tell the shell not to clobber files by setting the `noclobber` option.

```
$ set -o noclobber
$ ls > listing
bash: listing: cannot overwrite existing file
```

You might find this a useful option to set during the labs!

## Input Redirection

In a very similar way to `STDOUT`, shell commands which read from the *Standard In* file descriptor (`STDIN`) can read from a file using the *input redirector* <:

```
$ sort < listing
bar
bar
foo
foo
listing
```

# Standard Error

There is a third standard file descriptor, *Standard Error* or STDERR. This is not the same as STDOUT:

```
$ ls
foo
$ cat > baz < bar
bash: bar: No such file or directory
$ cat &> baz < bar
$ ls
baz    foo
$ cat baz
bash: bar: No such file or directory
```

Notice how the &> redirector sent STDERR to a file – in fact it sends STDOUT there too, so it’s very useful for debugging commands, i.e. failing X sessions:

```
$ startx &> /tmp/xerr        # N.B. Equivalent to: startx > /tmp/xerr 2>&1
$ less /tmp/xerr
```

# Some Special 'Files' in Linux

If we want to throw away output from a command we redirect it to `/dev/null`:

```
$ echo "now you see me, now you don't" > /dev/null
$ cat < /dev/null
$
```

Output to `/dev/null` is binned; attempts to read from `/dev/null` result in zero characters and an 'end of file' (`EOF`) marker.

`/dev` is a special directory in Linux, where *device nodes* exist. These look, and behave, like files, but they really interface down to hardware and other special devices, e.g. `/dev/hda` is the master disk on the first primary IDE channel, `/dev/fd0` is the first floppy drive, etc.

For programing, in addition to `/dev/null`, useful devices are:

- `/dev/zero` An infinite supply of *binary* 0s: `00000...`
- `/dev/random` A supply of cryptographically strong random bits: `1001110011101011101...`

## Pipes: Joining up commands

The concept of piping commands is fundamental to effective use of the unix shell. This is done with the pipe operator, `|`, which takes `STDOUT` from one command and gives it to `STDIN` of the next:

```
$ sort foo | tail -2
```

Will print the last 2 lines of `foo` sorted.

Pipes can be chained together

```
$ cat linux* | grep solar | tr [:lower:] [:upper:] | sort
```

Will print in upper case all the lines containing the word `solar` in all the files starting with `linux`, sorted in alphabetical order.

It's easy to see that commands like `head`, `tail` and `sort` are much more useful when used with pipes.

# Trickier Redirections

`STDIN`, `STDOUT` and `STDERR` are assigned file descriptor numbers 0, 1 and 2. These are used in more advanced redirections:

- `N> file` Redirect `N` to file
- `N< file` Redirect `N` from file
- `N>&M` Redirect `N` to descriptor `M`

Hence

```
$ frobnicate 2> errors 1> output 0< input
```

And

```
$ frobnicate 2> all_output 1>&2      # the same as "&> all_output"
```

Then

```
$ frobnicate 2>&1 | fankelise        # fankelise gets STDOUT and STDERR
```

# Job Control: Background Commands

Linux is obviously a multitasking kernel – it can run many processes at once (try `ps aux`).

The shell can use this feature to run more than one job at once: commands which end with & are run in the *background*.

```
$ frobnicate &
[1] 2099
$ echo -e "we can keep working\n while frobnicate runs" > /dev/null
$ echo -e "when it returns\n the shell will let us know..." >  /dev/null
[1]+ Done               frobnicate
$
```

If a job is started, and you want to subsequently put it into the background, stop it with `CTRL-Z`, then use the bg command. Bring it back with `fg` (foreground)

```
$ frobnicate                          # Type ^Z
[1]+  Stopped               frobnicate
$ bg
[1]+ frobnicate &
$ fg
frobnicate
```

# Job Control: Multiple Jobs

Each job has a job number: `jobs` prints these out:

```
$ jobs
[1]  Running                   frobnicate &
[2]- Running                   foo &
[3]+ Running                   bar &
```

Here 3 jobs are running. A job n can be brought out of the background with `fg %n`. Any stopped job n can be sent into the background using `bg %n`.

```
$ fg %1
frobnicate                                # Type ^Z
[1]+ Stopped                  frobnicate
$ jobs
[1]+ Stopped                  frobnicate
[2]  Running                  foo &
[3]- Running                  bar &
$ bg %1
[1]+ frobnicate &
```

The job descriptors `%+` and `%-` can be used to refer to the *current* or *previous* jobs.

# Job Control: Signals

Typing `^Z` sends a `signal` to the current foreground process – a `SIGSTOP`, in fact. There's another signal we can send with a keyboard shortcut, `^C` sends `SIGINT`. This will usually terminate the process, although some processes are immune (they *trap* or *catch* the signal).

General signals can be sent to any process by using the `kill` builtin (rather a misnomer – `SIGKILL` is only one signal, and it's not the default either).

```
kill: kill [-s sigspec | -n signum | -sigspec] [pid | job]... or kill -l [sigspec]
    Send the processes named by PID (or JOB) the signal SIGSPEC.  If
    SIGSPEC is not present, then SIGTERM is assumed.  An argument of '-l'
    lists the signal names; if arguments follow '-l' they are assumed to
    be signal numbers for which names should be listed.  Kill is a shell
    builtin for two reasons: it allows job IDs to be used instead of
    process IDs, and, if you have reached the limit on processes that
    you can create, you don't have to start a process to kill another one.
```

Specify a process using the shell job number, %n, or the *process identity number* (`pid`).

# Job Control: Process IDs and kill

To find out the `pid` use the `ps` command:

```
$ ps
  PID TTY          TIME CMD
  679 vc/1     00:00:01 bash
 1284 vc/1     00:03:20 frobnicate
 1285 vc/1     00:00:00 ps
$ kill -KILL 1284
[1]+  Killed          frobnicate
```

In this case we sent a `SIGKILL`, which is untrappable. Another common signal to send is `SIGHUP`, (hangup) which can ask a process to terminate nicely (e.g. to save its state) or reread a configuration file (a lot of system daemons respect this convention). Use `kill -l` to list all the avaliable signals.

# Unix System Information

As Linux is a multitasking system there may be several people logged into the system at once. It's important to be able to see what's going on in a Linux system (if only to see who's stealing your CPU cycles!).

The command `w` prints information about 'who's on and what are they doing':

```
$ w
 11:51:06 up 33 min,  3 users,  load average: 0.15, 0.10, 0.24
USER     TTY      FROM              LOGIN@   IDLE   JCPU   PCPU WHAT
graeme   vc/1     -                11:50   11.00s  0.15s  0.10s top
graeme   :0       -                11:18   ?xdm?  52.82s  0.00s -:0
fiona    pts/9    xi               10:20   15.00s  8.01s  0.00s -bash
```

Here there are three 'logins', one via X, one on a virtual console and another remote login from host `xi`.

Of particlar importance here is the *load average* information, which gives a good indication of how busy the machine is (the numbers are 1, 5 and 15 minute averages).

# Process Snapshots: ps

Of course `w` only prints information about users who have a current *login shell* – there are many other things going on on the system, which we can look at with `ps`.

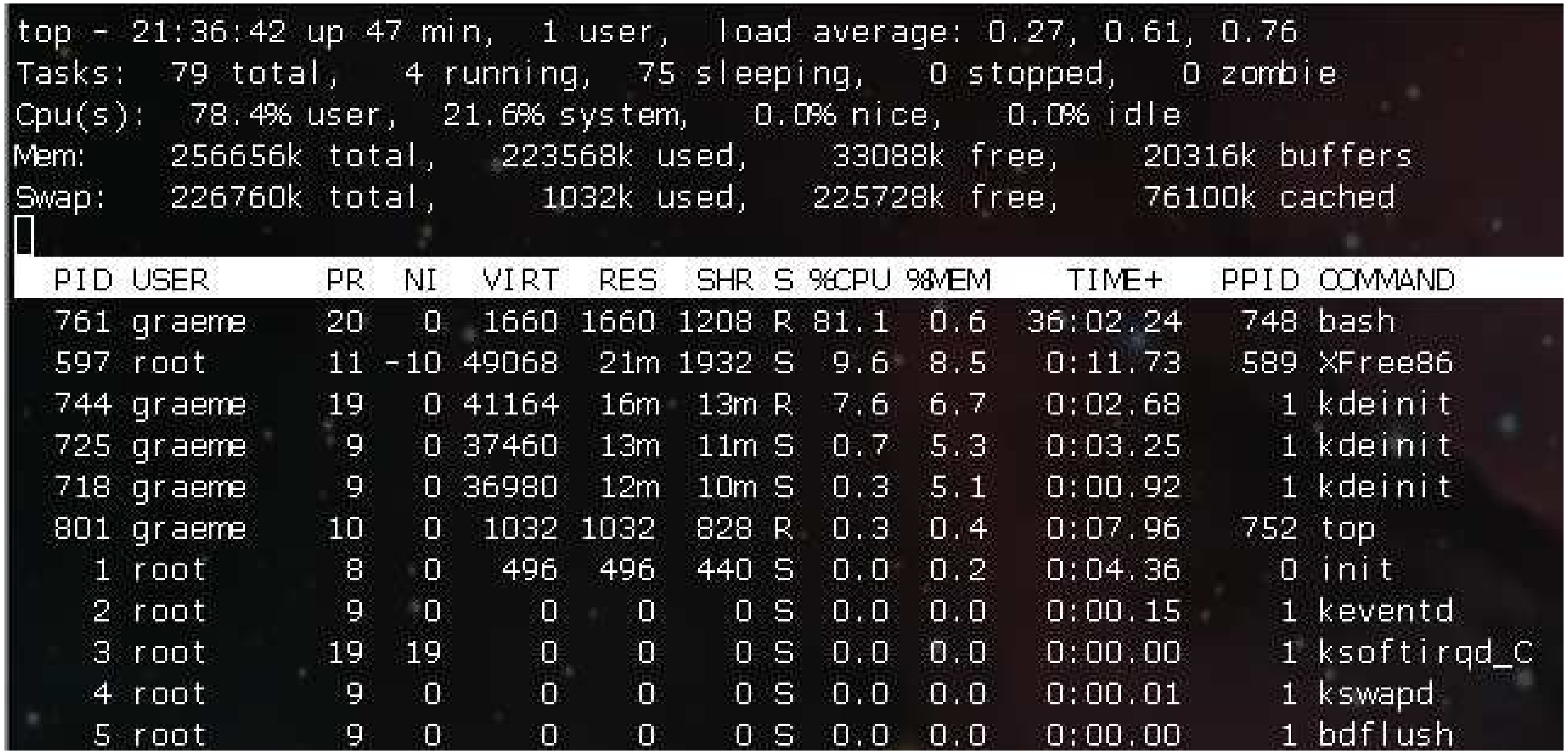
`ps`, without arguments, prints only your own process connected to the current terminal, but its output can be enhanced with various switches. (Oddly these switches do not take a “`-`” sign infront of them.)

| Option | Effect |
|---|---|
| x | Include processes with no controlling `tty` |
| a | Include all users, not just me |
| u | Long ‘user’ style listing |

`ps x` will print all your processes, `ps u` will print expanded information, `ps ax` will print all processes on the system. (`ps aux`: all process, expanded information.)

# Dynamic Display: top

ps gives a snapshot of processes, but to monitor processes in a continuous fashion use top:



```
top - 21:36:42 up 47 min,  1 user,  load average: 0.27, 0.61, 0.76
Tasks:  79 total,   4 running,  75 sleeping,   0 stopped,   0 zombie
Cpu(s):  78.4% user,  21.6% system,   0.0% nice,   0.0% idle
Mem:    256656k total,   223568k used,    33088k free,    20316k buffers
Swap:   226760k total,     1032k used,   225728k free,    76100k cached

  PID USER      PR  NI  VIRT  RES  SHR S %CPU %MEM    TIME+   PPID COMMAND
  761 graeme    20   0  1660 1660 1208 R 81.1  0.6  36:02.24   748 bash
  597 root      11 -10 49068  21m 1932 S  9.6  8.5   0:11.73   589 XFree86
  744 graeme    19   0 41164  16m  13m R  7.6  6.7   0:02.68     1 kdeinit
  725 graeme     9   0 37460  13m  11m S  0.7  5.3   0:03.25     1 kdeinit
  718 graeme     9   0 36980  12m  10m S  0.3  5.1   0:00.92     1 kdeinit
  801 graeme    10   0  1032 1032  828 R  0.3  0.4   0:07.96   752 top
    1 root       8   0   496  496  440 S  0.0  0.2   0:04.36     0 init
    2 root       9   0     0    0    0 S  0.0  0.0   0:00.15     1 keventd
    3 root      19  19     0    0    0 S  0.0  0.0   0:00.00     1 ksoftirqd_C
    4 root       9   0     0    0    0 S  0.0  0.0   0:00.01     1 kswapd
    5 root       9   0     0    0    0 S  0.0  0.0   0:00.00     1 bdflush
```

Press u to be able to select a single user’s processes; M to rank by memory useage rather than cpu.

## The XEmacs Text Editor

One of the original and best pieces of free software released under the GNU/FSF banner was Emacs. It started life as a set of macros for the TECO editor (in fact the name means *Editor MACroS*, although some have suggested more colourful interpretations).

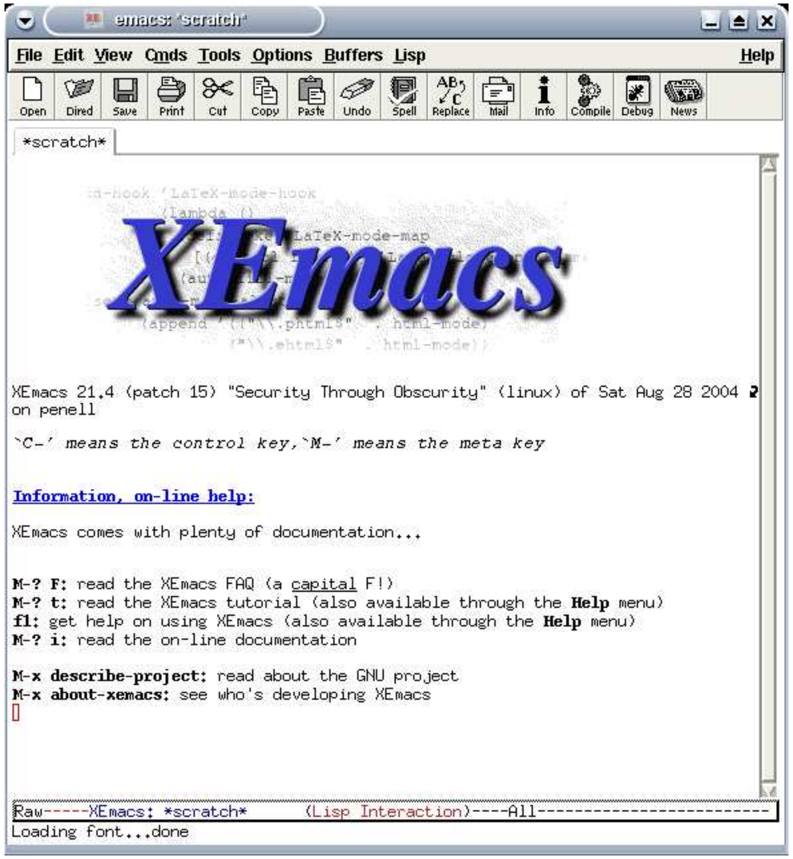
It quickly developed into a text editor in it's own right, but one which, because it could be programmed itself, could be customised to an incredible degree.

However, certain developers became frustrated with the direction of Emacs and, as it was free software, they forked the code to produce Lucid Emacs. The two emacsen developed in parallel for many years and Lucid Emacs became XEmacs, which we will use.

XEmacs is described as a "highly customizable open source text editor and application development system". It's an excellent text editor to use for shell scripting, C programming and many other development tasks.

# Starting XEmacs

To start XEmacs just type `xemacs` (or `xemacs &`) in the shell. You can also start it through the KDE menus on the cluster – naturally it's in the editors section.


emacs: *scratch*
File Edit View Cmds Tools Options Buffers Lisp Help
Open Dired Save Print Cut Copy Paste Undo Spell Replace Mail Info Compile Debug News
*scratch*
XEmacs

```
XEmacs 21.4 (patch 15) "Security Through Obscurity" (linux) of Sat Aug 28 2004
on penell

`C-' means the control key,`M-' means the meta key


Information, on-line help:

XEmacs comes with plenty of documentation...


M-? F: read the XEmacs FAQ (a capital F!)
M-? t: read the XEmacs tutorial (also available through the Help menu)
f1: get help on using XEmacs (also available through the Help menu)
M-? i: read the on-line documentation

M-x describe-project: read about the GNU project
M-x about-xemacs: see who's developing XEmacs

Raw-----XEmacs: *scratch*       (Lisp Interaction)----All-----------------------
Loading font...done
```

If you give the names of existing files on the command line XEmacs will start with those files opened for editing.

## XEmacs Keyboard Commands

XEmacs commands are usually activated by the `Ctrl` and `Meta` keys. The key sequence `C-x` means hold down the `Ctrl` key and press `x`. Similarly `M-x` means hold down the `Meta` key and press `x`.

However, PC keyboards do not have a `Meta` key – so the `Alt` key is used instead (but everyone still says ‘meta’!).

XEmacs has so many commands and functions that they cannot all possibly be activated by single key presses. Instead commands and functions can be run by being typed in the *mini-buffer*, activated by `M-x`.

So the command `M-x ispell-buffer` is run by

- Type `Meta-x` to activate the mini-buffer
- Type `ispell-buffer`

## Opening Files

To open a file in XEmacs use `C-x C-f`. This will activate the mini-buffer and the name of the file can be entered. Although this is not a point-and-click selection of files, XEmacs behaves very like the shell:

- it understands ~
- it offers tab completion
- and up-arrow recovers the last filenames entered

If the file does not yet exist it is opened for editing.

Once a file is opened in XEmacs it exists in an XEmacs *buffer* – so most commands will refer to *buffers* if they are acting on ‘opened files’. Commands which refer to ‘files’ are referring to the file system.

## Saving Buffers

To save a buffer (to a file!) use `C-x C-s`. XEmacs will *auto-save* a file every 300 key strokes or so. (The auto-save file for `foo.c` is `#foo.c#`.)

When you save a file with XEmacs the last version of the file is saved as a backup with a `~` appended, i.e., the backup file for `foo.c` is `foo.c~`.

Many file commands are avaliable through the *File* drop down menu, where the keyboard equivalent commands are given too.

# Working With Buffers

XEmacs can have many buffers open at once (note, not all of them will correspond to files). The *Buffers* drop down menu is the easiest way to switch between them, though you can also use `C-x b` to select a buffer and `C-x C-b` to list all the buffers.

To close a buffer, e.g., when you have finished editing a file but do not wish to exit XEmacs, use `C-x C-k`.

To split the XEmacs window in two type `C-x 2` – each window can then change buffer independently. Note that you can split windows as many times as you like, and that they can be resized by grabbing the dividing status bar with the left mouse button and dragging.

To close a window type `C-x 0`.

To make a window the only one displayed type `C-x 1`.

# Navigating Inside a Buffer

Within X it's really easy to navigate in XEmacs – a left mouse button click will move the text cursor. Also, the arrow keys, page up/down and scroll bar all work as you would expect (except on my laptop).

`C-HOME` and `C-END` move to the beginning or end of a buffer.

XEmacs can also navigate using command keys. These have some advantages – they generalise better, always work in weird terminals, and can be faster for experienced users. The disadvantage is that they do take longer to learn. However, for completeness, here are the most important ones:

| | | | |
|---|---|---|---|
| `C-f` | Forward char | `M-f` | Forward word |
| `C-b` | Backward char | `M-b` | Backward word |
| `C-p` | Previous line | `C-n` | Next line |
| `C-e` | End of line | `M-e` | End of 'sentence' |
| `C-a` | Beginning of line | `M-a` | Beginning of 'sentence' |
| `C-v` | Next page | `M-v` | Previous page |
| `M-<` | Start of buffer | `M->` | End of buffer |

# Kill and Yank in XEmacs

What would normally be called ‘cut and paste’ in most editors is called ‘kill and yank’ in XEmacs. (To be fair, XEmacs has been around a lot longer than most text editors, so one can forgive its slightly anachronistic syntax.)

The good news is that selecting text with the mouse in X enables the usual ‘cut’, ‘copy’ and ‘paste’ buttons on the toolbar.

However, the true power of ‘kill’ and ‘yank’ come (as usual) from using them via the keyboard:

- Basic killing is done with `C-k`. This kills from the cursor to the end of the current line. Keep pressing `C-k` and more lines are sucked up and killed.
- All of the killed lines go into the ‘kill ring’. To get the most recently killed item back, ‘yank’ it with `C-y`.
- After a ‘yank’ type `M-y` to cycle through the items in the kill ring – this enables one to select killed text from up to 16 kills ago.

## Kill and Yank on Regions

Using `C-k` to kill line after line is tedious on large regions. To kill arbitary areas:

- Set the ‘mark’ by typing `C-SPC`.
- Move where you want the region to end.
- Type `C-w` to kill the region or `M-w` to just copy the region to the kill ring.

Once a region is in the kill ring it can be yanked back as usual with `C-y`.

Many other XEmacs commands also operate on regions defined between the mark and the cursor – e.g., `M-x indent-region` will reindent, say, a piece of `C` code which has had a new control structure added.

# Searching in a Buffer

The search command in XEmacs is started with `C-s`. However, it's a little different from most editors in that it *incrementally* searches for text.

As you type in text XEmacs searches forwards for the closest match – and this changes as you make the search string longer. This is very useful to get away with the minimum typing.

- To jump to the next match type `C-s` again.
- If there is no match XEmacs says *failing I-search*.
- Press DELETE to remove characters from the match.
- To start searching backwards press `C-r` (you can switch between forwards and backwards searching).
- Once you are happy with the match, press ENTER.

## Exiting and Suspending XEmacs

To exit XEmacs type `C-x C-c`. You will be prompted as to whether to save any unsaved file buffers.

In a text console `^Z` (or `C-z` in XEmacs-speak) will suspend XEmacs, as normal.

In an X session `^Z` will minimise the XEmacs window (as it's usually a good idea to run XEmacs on a separate desktop this isn't hugely useful, in my opinion, but you may press `C-z` by mistake and wonder what happened to XEmacs!).

# Copyright