Linux Fundamentals

A Training Manual

Philip Carinhas, Ph.D.
Fortuitous Technologies
www.fortuitous.com
pac@fortuitous.com

Version of August 26, 2001

Contents

Chapter 1

Introduction to Linux

`Linux` is a computer operating system originally developed by Linus Torvalds as a research project. There is some interesting history about the rapid `Linux` evolution, but suffice it to say, `Linux` has come a long way in a decade.

`Linux` runs on Intel, Mac, Sun, Dec Alpha, and several other hardware platforms.

1.1 Linux Features

- `Linux` is a full-featured, 32-bit multi-user/multi-tasking OS.
- `Linux` adheres to the common (POSIX) standards for `UNIX` .
- Native TCP/IP support.
- A mature X Windows GUI interface.
- Complete development environment. C, C++, Java, editors, version control systems.
- Open Source.

1.2 Multi-User Operation

In `UNIX` and `Linux` , all interactions with the OS are done through designated “users”, who each have an identification ID (login name) and a password. `UNIX` allows different users to co-exist simultaneously and allows for different levels of users.

The most powerful user is called superuser or “root”, and has access to all files and processes. The superuser does many of the system management tasks like adding regular users, file backups, system configuration etc.

Common users accounts, which perform non-system type tasks, have restricted access to system-sensitive components to protect `Linux` from being accidentally or purposely damaged. In a moment you will enter a user account and start exploring the `Linux` filesystem.

1.3 Why Linux?

Linux can operate as a web, file, smb (WinNT), Novell, printer, ftp, mail, SQL, masquerading, firewall, and POP server to name but a few.

It can act as a graphics, C, C++, Java, Perl, Python, SQL, audio, video, and documentation, development workstation etc.

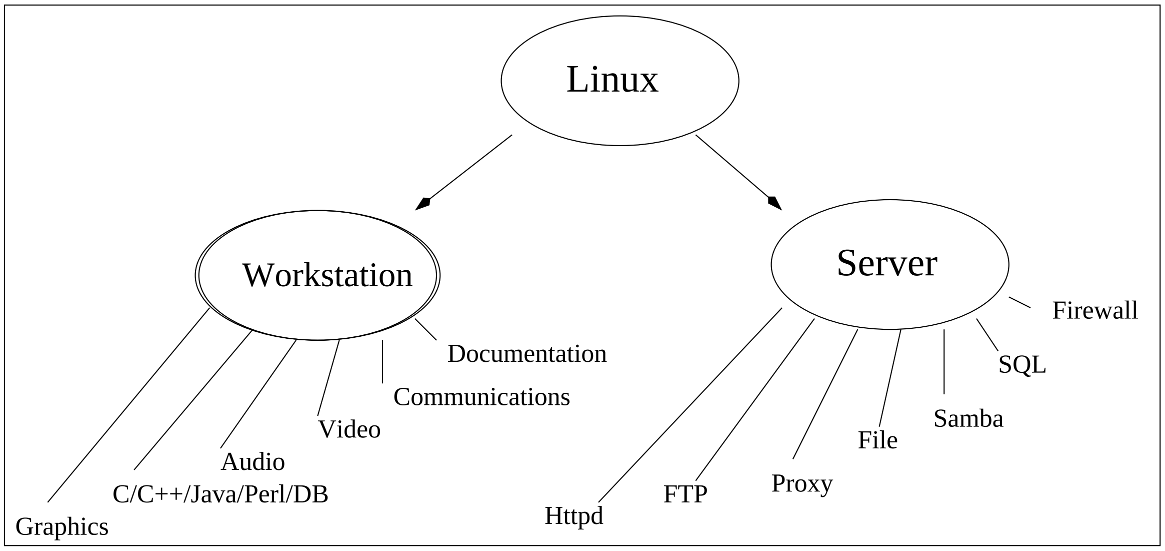


Figure 1.1: Linux Uses

Linux is a good solution for developers that need a stable and reliable platform that has open source code. Its not a good system for beginning developers that want a simple GUI interface to a programming language, although Linux has many GUI software development interfaces.

Linux is ideal as a workstation also, and offers many customizable features not found in any other platform. It makes a good platform for dedicated workstaions that have limited functions like in an educational or laboratory environment.

Its may not be ideal as a workstation for beginning users who want an instantly customizable universal WYSIWYG interface. Other systems provide solutions for this need. Still, Linux becomes easier to use on a daily basis. It's only a matter of time until Linux is accessible by everyone.

1.4 Conventions

In order to take full advantage of this manual, students should execute every command that is listed in the text as well as do all the exercises. The following is a list of conventions supported in this manual:

<bash>:	indicates a command entered in a terminal by the user. When you see this sign, you are expected to enter these commands exactly as indicated and check that the results are consistent with what is written. If you see a problem, please ask the instuctor to elaborate or clarify.
<tcsh>:	indicates commands in the `tcsh` shell. Most of these commands are to be completed after hours or at home. Since Linux advocates freedom of choice, we wish to make students aware of this option to `bash`.
<Super>:	indicates a command entered by the System Administrator or Root. The student is also expected to enter these commands in as indicated.
Bold	indicates a command entered at the prompts above.
`Big Bold TW`	is used to identify commands in the text.
`Underline`	indicates a file or a directory in the text.
`Slant`	indicates command options.
`Plain TW`	indicates a screen text of command output or editor.

Chapter 2

UNIX Command Line Basics

The objective of this chapter is to configure the shell account so that printing, manuals, and editor functions are working normally. This will give us experience with the basic commands, environment variables, and workhorse tools needed.

2.1 Logging In To Your Account

Log into your system with the login name and password given to you. You will see something like

```
<bash>:
```

Your system should always prompt you with the name of the shell (bash) and your login name. This is a customizable feature in the `bash` shell which you are now using.

The most basic command in `Linux` is the directory listing "ls" command.

You can see the contents of your account by typing

```
<bash>: ls -al
```

You should see files like

```
drwxr-xr-x  13 joe      1024 Nov  9 17:06 .dt
-rwxr-xr-x   1 joe      5111 Sep 30 15:19 .dtprofile
drwxr--r--   2 joe        96 Dec  1 12:25 .elm
-rwxr--r--   1 joe      1178 Nov 19 10:58 .emacs
```

2.2 Command Structure

We've already seen `ls` which give a directory listing the current working directory (cwd).

Commands in Linux follow a general format:

Command -*options* Other parameters

for example:

```
<bash>: ls    -l    .bashrc
```

First comes the command name, followed by options. Options are normally preceded by a dash or minus sign. There is always a space between the command and the dash. Some commands use no options at all. After the options comes any other parameters or informations that command may need.

Let's talk about some of the workhorse commands. Please note that these definitions are purposely abbreviated and incomplete! Consult the `man` pages for each of the commands below. In the commands below, parameters that are enclosed in square brackets [...] are optional to that command. [-opts] refers to options in the style just mentioned.

Special Keys Strokes:

q	quits from many commands like `more` and `less`
Ctrl+c	also quits out of many commands
Ctrl+L	clears the screen
Ctrl+a	puts the cursor at the beginning of the command line
Ctrl+e	puts the cursor at the end of the command line

Help, Search, Info Tools:

env *[-opts] [exp]*	Print environment or run a command with another environment.
find *[path] [exp]*	Find files in *path* using *exp*
info *keyword*	List *info* help pages containing *keyword*
locate *keyword*	Locate all files of name *keyword* in a database
man -k *keyword*	List *man* pages with *keyword* (same as `apropos` *keyword*)
man *command*	Display the manual for *command*
printenv	Print environment variables (see set)
set *[vars]*	Print/Set environment vars and functions
whatis *keyword*	Search the *whatis* DB for *keyword*
whereis *command*	Locate source/binary and manuals for *command*
which *command*	Display path of *command*

Text Manipulation Tools

awk| gawk *[pgrm][file]*	Filter *file* by *pgrm*
cat *file*	Display contents of *file* without paging
clear	Clears the screen. Same as Ctrl+L
grep *pattern file*	Finds *pattern* in *file*
head *file*	List the first few lines of *file*
more *file*	Display & page the text *file* (See `less`)
sed *[script] file*	Stream edit/filter *file* using *script*
tail *[-opts] file*	List the trailing lines of *file*
tr *chars1 chars2 file*	Change chars in *chars1* to *chars2*
less *file*	Display & page the text *file*

General Tools

cd *dir*	Change cwd to dir (home if dir omitted)
chmod *perms files*	Change file permissions of *files*
chown *owner.group files*	Change file owner and/or group
chsh	Change the default shell
cp *[-opts] f1 (f2/dir)*	Copy file f1 to f2 or directory *dir*
date	Displays the date
kill *pid*	Kills process ID *pid*
ln *[-opts] Old New*	Link *Old* to *New*
login *[username]*	Login to system with UID *username*
lpr *file*	Print *file* on the default printer
ls *[file]*	Listing for *file* (*cwd* if file omitted)
mkdir *dir*	Creates directory *dir*
mv *file1 file2*	Rename file1 to file2
passwd *[-opt] username*	Change password
ps *[-opts]*	Output a list of currently active processes
pwd	List the current working directory
rm *files*	Removes *files*
startx	Start the X-Windowing system
tar *[-opt][arch][file]*	Manage tar archives
telnet *[host [port]]*	Connect to the remote *host*
uname *[-opts]*	Output name and version number of OS
who	List users logged into this system
xterm *[-opts]*	Start a brand new X-terminal window

2.3 The Linux Manuals and the *man* Utility

Virtually every command that is worth knowing has an entry in the man pages, and is accessed by doing a `man command` . To get all related commands to a *keyword* word, use `man -k keyword` as in the following example:

```
<bash>: man -k manual
man (1)             - format and display the on-line manual pages
man2html (1)        - format a manual page in html
perlxs (1)          - XS language reference manual
whereis (1)         - locate the binary, source, and manual page files
xman (1x)           - Manual page display program for the X Window System
```

When in doubt, use `man` and `man -k keyword` to get info on a command or `UNIX` related term.. Find the man page for the `ls` command. `ls` will list directory contents:

<bash>: man ls

```
LS(1)                          FSF                          LS(1)
NAME
       ls - list directory contents
SYNOPSIS
       ls [OPTION]... [FILE]...
DESCRIPTION
       List information about the FILEs (the current directory by
       default).  Sort entries alphabetically if none of -cftuSUX
       nor --sort.
       -a, --all
              do not hide entries starting with .
...... etc ........
```

Figure 2.1: Manual Page for `ls`

Here is a `man` entry for `cd` which changes your current working directory (folder).

<bash>: man cd

```
cd(n)                 Tcl Built-In Commands                 cd(n)
NAME
       cd - Change working directory
SYNOPSIS
       cd dirName
DESCRIPTION
       Change the current working directory to dirName, or to the
       home directory (as specified in the HOME environment variable)
       if dirName is not given.  Returns an empty string.
```

Figure 2.2: Manual Page for `cd`

2.3.1 Exercises

1. **<bash>: man man**
2. **<bash>: man 7 signal**
3. Man `cd` and look for information on "." and ".."
4. **<bash>: cd .. ; pwd**
5. **<bash>: cd . ; pwd**
6. **<bash>: cd ; pwd**
7. Discusss what "." and ".." are in terms of the filesystem.

2.4 Create, List, Copy, and Move

The most fundamental of all commands is to list, create, copy, move (rename), and remove files. This section will show you the basics. Lets start with creating some files now, so we can list them later.

2.4.1 Creating New Files

File creation can be done by invoking an editor on a new filename. Before we try this, there is an easier way to make a new empty file by using the `touch` command. This is how it works:

```
<bash>: touch file1
```

This will make a new file named file1 that is empty. There are many other ways to make new files that we will see later.

2.4.2 Creating New Directories

Directories are created with the `mkdir` command. Thus

```
<bash>: mkdir dir1
```

will create a new directory named dir1 located in your current working directory. List this file now with the "long" format we just spoke about. Check your files with

```
<bash>: ls -l
```

2.4.3 Listing Files and Directories

By far, listing a file is the most basic of all commands. As we have seen before, you list files with the `ls` command:

```
<bash>: ls file1
  file1
<bash>: ls -l file1
  -rw-r--r--    1 joe   users      0 Oct 28 22:05 file1
```

Using the -l flag causes a "long" listing that shows more information about file1.

You list your directories in a similar way (note that the second example shows that dir1 is empty).:

```
<bash>: ls -l
  total 10
  drw-r--r--    1 joe   users      0 Oct 28 22:05 dir1
<bash>: ls -l dir1
  total 0
```

2.4.4 Copying Files and Directories

You can copy a file by using the `cp` command. The following statement

```
<bash>: cp file1 file2
```

will copy `file1` to `file2`. Copying a folder or directory requires the use of the *recursive* or -r flag indicating that `cp` should decend into the directory and copy all sub-files and sub-folders:

```
<bash>: cp -r dir1 dir2
```

Exercise 2.4.4: Please create a directory like this now. Explain clearly how the following examples are different from the above:

```
<bash>: cp file1 dir1
<bash>: cp -r dir1 dir2/
<bash>: cp -r dir1 dir2/dir3
```

2.4.5 Moving Files and Directories

Moving a file is the same as renaming it. This allows for the possiblity that you move the file to another directory as well:

```
<bash>: mv file2 file3
<bash>: mv file1 dir1/file2
```

You move a directory in EXACTLY the same way as a file:

```
<bash>: mv dir1 dir3
```

2.4.6 Changing Directories

Changing your current directory is done with the `cd` command:

```
<bash>: cd dir1
<bash>: cd ../dir2
```

2.4.7 Removing Files and Directories

Normal files are removed with the `rm` command:

```
<bash>: rm file1
```

Removing directories is also done with the `rm` command, but again you need to use the "recursive" or -r option:

```
<bash>: rm -r dir1
```

After you try the above, make sure `file1` and `dir1` are gone.

2.5 I/O, Redirection, and Pipes

I/O refers to Input (I) and Output (O). This section talks about input and output of commands and how you can manipulate these. These prinicipals are very important because they are used constantly in Unix.

2.5.1 Standard I/O

In `UNIX` , Standard Input (*stdin*) and Standard Output (*stdout*) are mechanisms that allow you to input or output data from a command line. Simple commands like "`cat file1`" send their results to *stdout* (normally to your terminal screen) while the word `file1` is an example of *stdin* which is fed to the command `cat`.

Independent of *stdin* and *stdout*, there is the also standard error (*stderr*) which normally goes to your screen when the command detects an error. Its manipulation is shell specific.

Bash assigns special numbers, called *File Descriptors*, to *stdin*, *stdout*, and *stderr*:

Name	Abbreviation	File Descriptor	Standard Device
Standard Input	*stdin*	0	Keyboard
Standard Output	*stdout*	1	Console
Standard Error	*stderr*	2	Console

Table 2.1: Bash Standard I/O

2.5.2 Redirection

Redirection refers to the art of redirecting input and output traffic from commands. Shells like `bash` allow for redirection of *stdin* and *stdout* with the `<` and `>` operators respectively.

As an example, lets say you want to list some files and send (redirect) the output to a file instead of the screen. Do it the easy way:

```
<bash>: ls -al > output.txt
```

To check the output, you can use `cat` (short for concatenate). `cat` is uselful when you want to view short files:

```
<bash>: cat output.txt
-rwx------   1 carinhas users        1606 Aug 17 23:14 .acrorc
-rwx------   1 carinhas users         153 Dec 20 08:47 .bashrc
-rwx------   1 carinhas users        3189 Dec 23 15:34 .cshrc
................
```

Examples of *stdin* redirecting:

```
<bash>: cat < output.txt
<bash>: wc -l output.txt
    7 output.txt
<bash>: wc -l < output.txt
    7
```

Note that "`cat output.txt`" and "`cat < output.txt`" give the same result, but the "`wc -l`" examples give something slightly different.

In `bash` *stderr* is redirected with with the `2>` operator, while in `tcsh`, the `>&` operator. Just relax and we will see real examples of this shortly.

Linux Warning: Please note that `>` will overwrite anything in the output file, if it exists, or create the file if it does not exist. In contrast, the `>>` operator will append to the existing file.

Here is a brief summary of the redirects:

Name	Operator	Description
Redirect *stdin*	<	Feeds the file to input
Redirect *stdout*	>	Creates or overwrites
Append *stdout*	>>	Creates or appends
Redirect *stderr*	>&	Both *stdout* and *stderr*
Redirect *stderr*	2>	Only *stderr* in bash

Table 2.2: Standard Redirection I/O

2.5.3 Pipes

When you want to take the output of one command and use that is input into another, use the "pipe operator" `|` . Think of actually connecting a metal pipe from one command to another. The following example sorts a simple `ls` command in reverse order (do it!):

```
<bash>: ls |  sort -r
 .doomrc
 .cshrc
 .bashrc
 .acrorc
```

2.5.4 Examples to Try

```
<bash>: cat noname 2> error.txt        # Sends error to error.txt
<bash>: cat noname > error.txt 2>&1    # Send stdout + stderr to error.txt
<bash>: cat noname >& error.txt        # everything (as above) goes to error.txt
```

Try these examples in tcsh:

```
<tcsh>: tcsh
<tcsh>: (cat noname > output.txt) >& error.txt    # Send stderr to error.txt
<tcsh>: more output.txt                           # Same as in bash.
<tcsh>: more error.txt                            # Just like in bash.
<tcsh>: cat testfile.txt >& out.txt               # ditto
<tcsh>: cat < testfile.txt | sort | more          # ditto
<tcsh>: locate .bashrc | xargs grep alias         # find 'alias' in .bashrc.
<tcsh>: exit
```

Don't forget the last "exit" to get out of `tcsh` and back into fabulous `bash`.

2.6 Command Line Editing

A great shortcut in `bash` and `tcsh` is to use the command editing facilities. Just hit the up-arrow key to a previous command and move the cursor to edit that command. Just try it. Command editing is very useful in repeating and correcting previous commands.

Now we discuss other features of command line editing.

2.6.1 Command and File Completion

Most shells support command and file completion typing shortcuts. These shortcuts allow you to hit the Tab key to finish off the name of a command or file after only hitting a few keys. Try this example of a filename (directory):

<bash>: **cd /usr/inc<TAB>** will produce
<bash>: **cd /usr/include/**

Now try this example for a command:

<bash>: **ghostv<TAB>** should produce
<bash>: **ghostview**

Note that command and file completion can only complete upto a unique set of commands. This means you have to provide a unique start string.

Thus <bash>: **ghost<TAB>** won't work because it could complete to `ghostview` or `ghostscript`.

`tcsh` has the same file-name completion mechanism as bash. The <TAB> key will complete the filename and command up to unique names.

2.6.2 Possible Command Completion

`bash` will show the possible choices are by hitting <TAB>twice (<Ctrl-D> in `tcsh`):

```
<bash>: ls /etc/h<TAB><TAB>
host.conf    hosts        hosts.allow  hosts.deny   httpd/
<bash>: ls /etc/h
```

The shell tells you what your choices are and is again ready for more input. This also works on commands too:

```
<bash>: mo<TAB><TAB>
modemlights_applet      montage                  mount.smbfs
modemtool               more                     mouse-properties-capplet
modinfo                 morepgp                  mouse-test
modprobe                mount                    mouseconfig
<bash>: mo
```

This shows us the possibilities, and again returns us so we can continue typing a command.

Remember that `tcsh` has the same possible-completion mechanism as `bash` does above. Just use <Ctrl-D> instead.

2.6.3 Command Line Substitution and History

In both `tcsh` and `bash` we have the facility of *Command Line Substitution* and *Command History*.

Command Line Substitution allows you to substitute a dynamic expression inside a command:

```
<bash>: echo My name is $USER
My name is Joe
<bash>:
```

The $ symbol allows bash to process the expression in-line and later provide the results to the `echo` command.

Command History allows you to use stuff from your old commands in your current command. Remember: recycling is good for the environment ♣. The following list shows the history reference syntax common to both `bash` and `tcsh`:

!!	**Redo the last command. Same as `!-1`**
!−*N*	**Repeat the N^{th} most recent command (see next)**
!−3	**Repeat the 3^{rd} most recent command**
!*N*	**Redo the N^{th} entry in the history list**
!string	**Redo last command starting with the text "string"**
!?string?	**The most recent command which contains the text "string"**

Figure 2.3: History Referencing Specifications

On top of *history referencing*, you can add these modifiers to the shell command line by appending them to the history reference after a colon (:)

`0`	`The first (command) word`
`n`	`The nth word on the command line besides the command.`
`^`	`The first word, equivalent to '1'`
`$`	`The last word`
`%`	`The word matched by an ?s? search`
`x-y`	`A range of words.  '-y' abbreviates '0-y'.`
`*`	`Equivalent  to  '^-$',  but returns nothing if the event contains only 1 word`
`x*`	`Equivalent to 'x-$'`
`x-`	`Equivalent to 'x*', but omitting the last word ('$')`

Figure 2.4: History Modifiers

2.6.4 Exercises

You can get a list of your history by typing simply `history`. Create two files called `boogie.man` and `boogie.man.old`. The file contents are not important for this exercise. Try these examples of the above history machinery to try. The exact history will vary depending on what you do so please adapt these to your current situation:

```
<bash>: history |  tail -4
      9  8:30    touch boogie.man
     10  8:31    cp boogie.man boogie.man.old
     11  8:36    echo hello >> boogie.man
     12  8:37    diff boogie.man.old boogie.man
```

```
<bash>: !-2
vi boogie.man
```

```
<bash>: diff !-2:2.old !-2:2
diff boogie.man.old boogie.man
```

```
<bash>: diff !!:1.old !!:2
diff boogie.man.old.old boogie.man
diff: boogie.man.old.old: No such file or directory
```

```
<bash>: !?cat?:0 !!:2
cat boogie.man
```

Chapter 3

The Linux Environment

In Linux and UNIX , you carry many local attributes that control how your immediate shell interface interacts with the system. These attributes are called *Shell and Environment Variables*, and they are critical to Linux operation. UNIX and Linux use many variables for internal processing. These help the system manage resources for users. This chapter discusses the bare necessities that all Linux survivalists need to know.

3.1 The UNIX Shell Game

One prominent feature of UNIX is that you can control the interface by which you communicate with it. These interfaces are called *s*hells and there are too many. The original shell is called sh and it has many close cousins including ash, bash, and ksh. bash is the "native" Linux shell and is the default shell for Linux . Thus, bash is used for almost all startup files (scripts) and maintenance scripts for Linux .

Many users prefer to use the Berkeley UNIX C-type shells like csh and tcsh. Your choice is largely a matter of preference, however, we will use bash as the default in the course. We will also discuss tcsh because of its easy to understand C-like syntax.

Normally a shell can be invoked anywhere in your current login session simply by typing the shell name. This is useful for testing, but normally, you will use your default shell.

A user can control fundamental aspects of his/her environment by setting shell variables as you work, or inside of a startup script file which gets invoked when you login to your account. Simple examples common to each shell are your prompt (the thing that stares at you and waits for your commands) and aliases. An alias is a shorthand for some other command which a user can define, normally in the shell startup files like *.bashrc* and *.cshrc*. We'll work with these concepts very shortly.

3.1.1 Shell Features

Most `Linux` distributions come with `bash` and `tcsh` which replace `sh` and `csh` respectively. These are the two big shells that 99.9% of the Linux world uses. There are several key commonalities and differences between the two. The following tables outline these:

Action	bash	tcsh
Default Prompt	$	#
Force Redirection	>|	>!
Force Append		>>!
Variable assignment	var=value	set var=value
Set Environment var	export var	setenv VAR value
Number of arguments	$#	$# ($#argv in csh)
End loop statement	done	end
End case statement	esac	endsw
Loop	for/do	foreach
If statement	if [...]	if (...)
End if	fi	endif
Read from terminal	read	$<
Start until loop	until/do	until
While loop	while/do	while

Table 3.1: Bash-Tcsh Difference

Symbol/Command	Action
>	Redirect output
>&	Redirect *stdout* and *stderr*
>>	Append output to file
|	Pipe output
&	Run process in background
TAB	Filename/command completion
!n	Repeat command #n in history
$var	Variable substitution
#	Comment line
bg	Put process in background
fg	Put process in foreground
cd	Change directory
echo	Echo execution output
jobs	List current shell processes
kill	Kill specified processes
umask	Set default file permissions

Table 3.2: Bash-Tcsh Commonality List (Incomplete)

3.2 Bash

Bash is the default shell for `Linux` . The default shell is like an environment where all the basic commands come from bash. Let's look at the bash manual page:

```
<bash>: man bash
```

Look at the default prompt on your screen. Notice that the prompt contains the shell name and username. The prompt in `bash` is controlled by the variable *PS1*. Type in the command "set" to get the following sort of output:

```
PRINTER=astro
PS1=<\s>:
PS2=>
PS4=+
PWD=/home/joe
SHELL=/bin/bash
SHLVL=2
TERM=vt100
```

3.2.1 Setting Your Prompt

Note that `set` provides the definition for *PS1* and several other variables. Lets look up the manual on `bash` by typing "`man bash`". You are looking at an endless stream of information from the native `Linux` manual pages. Type "*/prompt*" to search for the word "prompt". You can find the next occurrence by typing "*n*". Keep going until you see something similar to

```
PROMPTING
       When  executing  interactively,  bash displays the primary
       prompt PS1 when it is ready to read  a  command,  and  the
       secondary  prompt PS2 when it needs more input to complete
       a command.

              \t     the current time in HH:MM:SS format
              \d     the date  in  "Weekday  Month  Date"  format
              \s     the  name  of  the shell, the basename of $0
              \w     the current working directory
              \W     the basename of the current  working  directory
              \u     the username of the current user
              \h     the hostname
              \#     the command number of this command
              \!     the history number of this command
              \$     if  the effective UID is 0, a #, otherwise a $
```

Lets change the `bash` prompt by adding the command number. This is an example:

```
<bash>: PS1=' \s: \u \t:# \#: '
```

which should produce something like

```
bash: joe 00:02:07:# 13:
```

Now lets set your prompt so that it always displays your path. Even though this manual will specify `<bash>:` , you should still use a prompt like this:

```
<bash>: PS1='<\w>: '
```

3.2.2 Creating Aliases (Shortcuts)

Now as it happens, I get very tired of typing out `ls -Agl` every time I want a long directory listing, so I want to alias that command to `d` (for "directory") as follows:

```
<bash>: alias d='ls -Agl'
```

We really would like to make this definition permanent, by putting this command into `.bashrc`, and we will learn how to do this in section 7 on scripts.

The same `alias` in `tcsh` (or `csh`) is written without the equals (=) sign: You start `tcsh` by typing *tcsh*:

```
<bash>: tcsh
<tcsh>: alias d 'ls -Agl'
<tcsh>: d
 .............
<tcsh>: exit
```

3.3 Shell Variables

In `Linux` , you always keep shell variables *user*, *directory*, *path*, *group*, and many others. These variables help your shell cope with all the information it needs to deal with normal operations. As we discussed before, the *set, env, printenv* commands will display your environment variables: (without #comments)

```
PWD    = /home/carinhas/misc  # Your current working directory
GID    = 100                  # Your current group ID number
GROUP  = users                # Your current group name
HOME   = /home/joe            # Your home directory
HOST   = astro                # The name of your workstation.
PATH   = /bin:/usr/bin        # Your current path to executable commands.
PS1    = <\s>:                # Your current prompt
SHELL  = /bin/bash            # Your current shell
UID    = 501                  # Your user ID number
USER   = carinhas             # Your user name
```

Your *current working directory* (*cwd*) is where all files are sought and written by default. *uid* and *gid* are the numerical values for your *user* and *group*, which identify you as a user and group member respectively. Your *home* is where you (and your files) live, and your *path* is where you look for your prey (executable commands).

You can always determine your *cwd* with the `pwd` command. Your *cwd* is always referred by the "dot" (.) symbol, and the directory directly below is referred to by the double-dot (..) symbol as in the result of an `ls -Agl`:

```
<bash>: ls -gl
total 608
drwxr-xr-x   2 joe      users        1024 Dec 20 19:09 .
drwxr-xr-x  15 root     root         1024 Dec 10 19:04 ..
-rwx------   1 joe      users        3185 Dec 19 15:51 .cshrc
-rwx------   1 joe      users        3185 Dec 19 15:51 .bashrc
-rw-r--r--   1 joe      users          33 Dec 20 10:49 doubletake
drwxrwxrwx   1 joe      users          33 Dec 20 10:49 mail
```

Note that ".." refers to `/home` when you are in your home directory `/home/joe`. The first character in the long file listing is the *file type*, which tell us what kind of animal we have. Also ~ refers to your home directory in your prompt, and changes to the `/fully/qualified/path` only when moved out of `/home/joe` .

3.3.1 Environment Variables

Environment variables are shell variables that control the so-called '*environment*', or shell options. *Environment* variables are used internally by the shell at all times. Normally, user defined shell variables will not affect how the shell works.

Environment variables are all uppercase words like *PATH, SHELL, TERM, PAGER, PS1, MANPATH.*

In `bash`, you set *environment* vars with the `export`:

```
NAME=value
export NAME
```

Or you can do it all in one command:

```
export NAME=value
```

Examples in *bash* are

```
<bash>: export PATH=/bin:/sbin:/usr/sbin
<bash>: DISPLAY=groucho:0
<bash>: EDITOR=/bin/vi
<bash>: export DISPLAY EDITOR
```

Environment variables are set in `tcsh` in the following way:

```
setenv NAME value
```

3.3.2 Exercises

Exercise 1

Find the hostname of your machine and replace "groucho" in the DISPLAY variable as above.

Exercise 2

Go into the `tcsh` shell and work the following examples:

```
<bash>: tcsh
<tcsh>: setenv PATH /bin:/sbin:/usr/bin:/usr/sbin
<tcsh>: setenv DISPLAY groucho:0
<tcsh>: setenv EDITOR /bin/vi
<tcsh>: env
```

3.4 Choosing the Right Path

Your *PATH* is used by the system to find commands. Its important to set this correctly so that you can access everything that you need.

To set you path in `bash` we edit the `.bash_profile` file:

```
<bash>: pico .bash_profile
PATH=$PATH:$HOME/bin                    # Add $HOME/bin to $PATH
.... write and quit .... (hit Ctrl-X, type Y, return)
<bash>: source .bash_profile        # Initialize $PATH.
```

Once this is done, `bash` will look in your own `/home/user/bin` folder to find any commands you put there. This is a good thing which will come in handy later when we work on scripting.

3.4.1 Exercises

Set your path in `tcsh` by editing your `.login` file, and add `~/bin` to your path.

```
<tcsh>: pico .login
set path = (/bin /sbin ~/bin /usr/bin /usr/ucb /usr/bsd )    # added ~/bin
.... write and quit ....
<tcsh>: source .login                    # Install the new $path.
<tcsh>: rehash                           # Tell csh to find files in $path.
```

`tcsh` uses its `rehash` function to re-read all the folders in your path and find the new commands.

3.5 Groups and Newgrp

Groups allow users to have access to files and programs jointly. This is what the *group* attribute is for. Users can belong to many groups. You can belong to more than one group, however you can only be active in one group at a time. All users have a default group that they enter at login. You change your group interactively by using the `newgrp` command:

```
<bash>: newgrp groupname
```

which will spawn a new shell (on top of the old one) with your default group given by *groupname*. For this trick to work, you must belong to this group to begin with.

Once you are in this new *group state*, all process and files created will carry the *group* name as well as your *user* name. Having multiple groups can be useful when accessing restricted hardware or working in a team of developers, where everyone needs access to common software and systems.

To exit this group-shell, simply type `exit`.

3.5.1 Exercises

1. Look at the files `/etc/passwd` and `/etc/group`. Determine how groups are created and assigned with the help of the manual pages. Write your observations down here.

Chapter 4

Filesystem Essentials

In this chapter, we will learn how to master files and filesystems. This is a very important aspect of `Linux` , and is used constantly.

4.1 The Linux Virtual Filesystem (VFS)

All files in Linux are accessible from a single integrated filesystem called a *Virtual Filesystem*. The base of that system is called the *root* and often denoted by a slash (/). Use the `mc` command to navigate around the system files below:

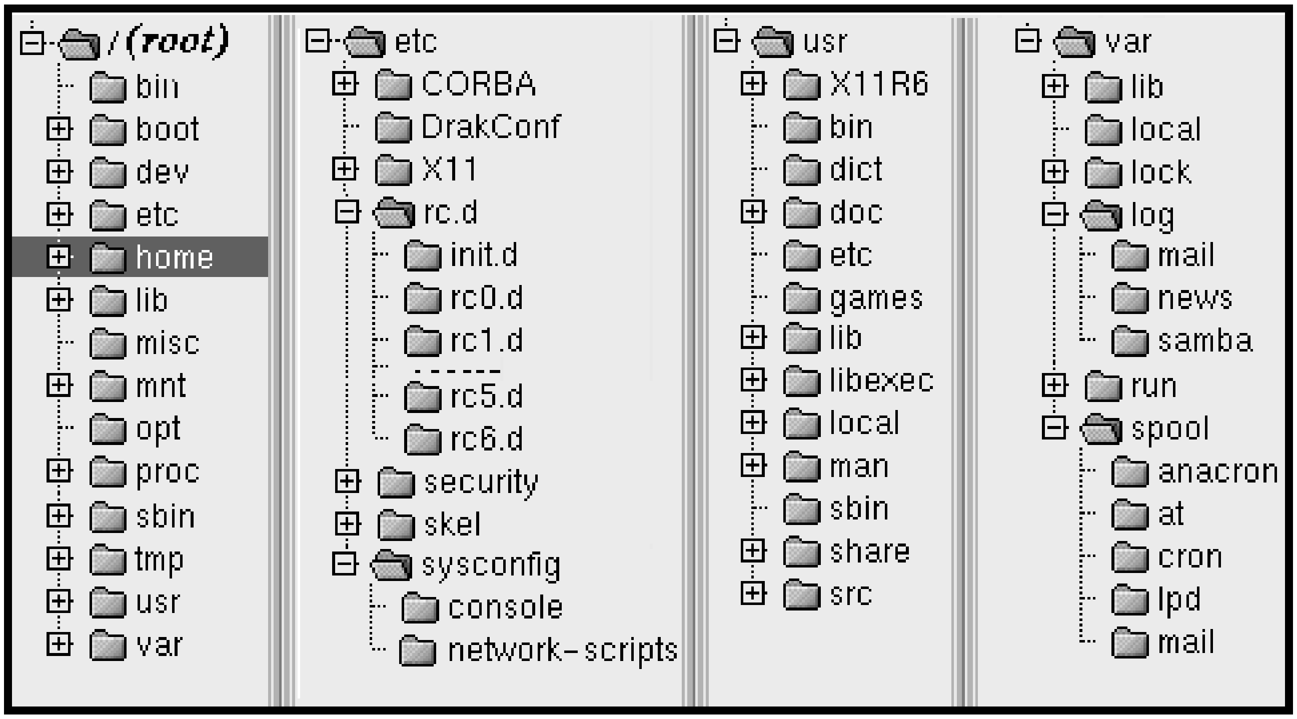
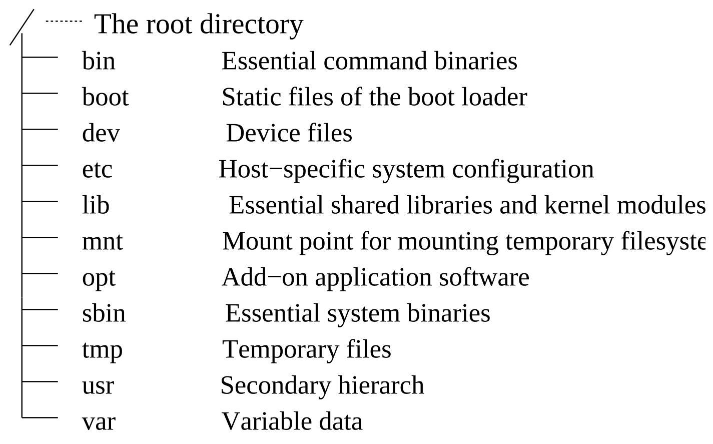


Figure 4.1: Linux VFS as viewed by `mc`

The next figure outlines the root VFS root tree as set by the Filesystem Hierarchy Standard located at `http://www.pathname.com/fhs/`



/	The root directory
bin	Essential command binaries
boot	Static files of the boot loader
dev	Device files
etc	Host−specific system configuration
lib	Essential shared libraries and kernel modules
mnt	Mount point for mounting temporary filesyste
opt	Add−on application software
sbin	Essential system binaries
tmp	Temporary files
usr	Secondary hierarch
var	Variable data

Figure 4.2: VFS Root Tree

Among the most important of these are `/`, `/etc`, `/usr`, and `/opt`, `/tmp` and `/home`. Below is a sample directory tree that starts from root but lacks the `/var` subtree.

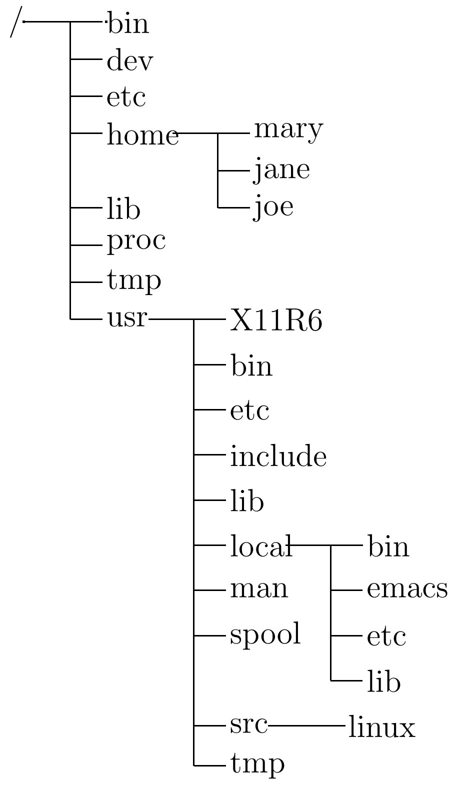


Figure 4.3: A Linux directory tree.

4.1.1 Exercises

1. The tree diagram above is not correct for a modern version of Linux. Find and compare the differences with your system and write down your results.

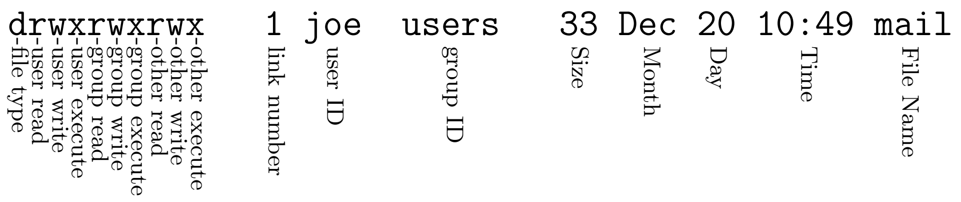
2. The `/proc` filesystem is a special on that has direct access to the system kernel. The files in `/proc` have no size, but what happens if you do the following:

   ```
   <bash>: cat /proc/cpuinfo
   <bash>: cat /proc/pci
   ```

   Discuss the possible uses of this.

4.2 File Attributes

Lets take a close look at a directory listing of *mail.* Enter the command `ls -agl mail`:



```
drwxrwxrwx   1 joe  users   33 Dec 20 10:49 mail
```

Figure 4.4: Unix File Attributes

Its entry gives us a lot of information. It has all of its access entirely open, so everyone can read, write, and execute. Also note that the first attribute listed is its type, in this case a “d” for directory. File attributes for restricted (plain type) files are listed as dashs (-).

The next 9 parameters are *file permissions* or *access parameters* In `UNIX` , files have 3 sets of 3 access parameters or “permissions”. The first 3 are for the owner, the second 3 for the group, and the last 3 for “other” or the “world”. The permissions are *read* (r), *write* (w), and *execute* (x). When you look at a file listing in long format you will see these attributes listed as (r,w,x) or omitted with a “-” in their place. If the attributes are present, access is granted to for that action, otherwise access is denied.

Other major file types include symbolic links (l), an internal reference to another file, and plain files (-). The following table summarizes native `Linux` file types as listed by the `find` command. Note that the plain file permission never lists by `ls` as (f) but as a dash (-).

b	block special file
c	character special file
d	directory
l	symbolic link
p	named pipe (FIFO)
s	socket

Table 4.1: File Types as Listed by `find`

4.3 Changing File Attributes with `chmod`

An important task for any `Linux` user is to manage file attributes (permissions) with the `chmod` command. The syntax for `chmod` is

```
chmod  [-R]  MODE  FILE
```

where MODE can be a comma separated combinations of 3 sets of symbols:

```
[ugoa][+ - =][rwx]

 u --> user
 g --> group
 o --> others
 a --> all
```

You can add (+), subtract (–), or exactly set (=) the mode. Examples:

```
 chmod u+w   mail   # Add user (u) write (w) privileges
 chmod u-w   mail   # Remove user (u) write privileges.
 chmod g+w   mail   # Add group (g) write privileges
 chmod g=r   mail   # Allow only the group read privileges, nothing else.
 chmod o+x   mail   # Add execute (x) privileges for others.
 chmod a+x   mail   # Add execute (x) privileges for everyone..
 chmod a=xr  mail   # Allow read and execute only to everyone..
 chmod go-r  mail   # Remove group and others read privileges..
 chmod ugo+w mail   # Same as chmod a+w mail
```

MODE can also be an *octal* representation which represents the absolute mode:

```
  1 = others execute # o=x
  2 = others write   # o=w
  4 = others read    # o=r
 10 = group execute  # g=x
 20 = group write    # g=w
 40 = group read     # g=r
100 = user execute   # u=x
200 = user write     # u=w
400 = user read      # u=r
```

The trick with *octal* is that you can add up the different numbers to get all different combinations of privileges like:

```
  1 = others execute # o=x
+ 2 = others write   # o=w
+ 4 = others read    # o=r
-----------------------------
= 7 = other read, write, and execute = o+rwx

 10 = group execute  # g=x
 20 = group write    # g=w
 00 = no group read  #
100 = user execute   # u=x
000 = no user write  #
400 = user read      # u=r
-------------------------- add these last 6 to get
530 = u=rx, g=wx

chmod 750 mail => chmod  u=rwx,g=rx,u=  mail
chmod 700 mail => chmod  u=rwx,g=,u=    mail
chmod 751 mail => chmod  u=rwx,g=rx,o=x mail
chmod 741 mail => chmod  u=rwx,g=r,o=x  mail
```

4.3.1 Exercises

1. On directories, the execution privilege allows for access into the directory, so having just read and write (rw) privilege is not enough. Only execute privilege will allow any access to anything inside the directory. Try this out by doing a

   ```
   <bash>: chmod 600 mail
   <bash>: cd mail
   ```

   There are actually many other modes that can be set using the first (non-octal) method. The ones we discussed are the essential ones. Please check out the `man` pages on `chmod` and `ls`.

2. Restore your mail directory back to a usable mode.

3. Write down the command you just used in both octal and symbolic modes.

4. Read the man pages and learn how to set and the function of the "set user ID on execution" (SUID) bit for `chmod`. Discuss this with your classmates and instructor. This concept is a requirement for the LPI exams.

4.4 Changing File Ownership with `chown` and `chgrp`

Just as you can change the file permission, you can change the *owner* and *group* of a file. `chown` does this for you with the syntax:

```
chown  [-R] USER[.[GROUP]] FILE
```

Note that `chgrp` can change the *group* at the same time as in the following examples.

```
<bash>: chown johnson mail         # Sorry, only johnson or Root can do that.
chown: mail: Operation not permitted
<Super>: chown joe.users mail    # Change owner to joe and group to users.
<Super>: chown joe.bozos mail    # Change owner to joe and group to bozos.
chown: mail: Operation not permitted       # You're not a member of bozos group!
<Super>: chown .testers mail       # Ok, you are in the testers group.
```

`chgrp` is less powerful and not really needed but should be noted. Its syntax is

```
chgrp [-R] GROUP FILE
```

Here is an example.

```
<bash>: chgrp testers mail   # Same as ``chown .testers mail''
<bash>: chgrp users   mail   # Same as ``chown .users  mail''
```

4.4.1 Questions

Why can't you change ownership of someone elses files? Who can do this? Is this a good or bad feature?

4.4.2 Exercises

1. Log in as root and change the owner and group of some files in the `/tmp` directory. Make sure you are in `/tmp` and not `/home/you/tmp` or some other place.

4.5 Devices

In Unix (almost) all commands and hardware are represented by files. The hardware devices are fundamentally different and live in the directory `/dev`. Device files allow access to the hardware directly through the kernel.

These special files have *major* and *minor* numbers that identify them in the `Linux` kernel. *Major Device Numbers* specify a particular driver for I/O redirection in the kernel, and *Minor Device Numbers* specify a device to be accessed by that driver. A typical example is the `/dev/hda` hard-drive:

```
brw-rw----   1 root     disk       3,   0 May  5  1998 /dev/hda
```

It has a major number 3 and minor number 0. The leading "b" indicates that this is a *block* device. It's owner is **root** and its group is **disk** .

Do an `ls -agl /dev` to see the major and minor numbers of typical devices.

We list the most commonly used devices below. Note that N will represent an integer number.

Device Name	Type	**Description**
/dev/hda	B	first IDE hard disk
/dev/hda$N$	B	Nth partition of /dev/hda
/dev/sda	B	first SCSI hard disk
/dev/sda$N$	B	N^{th} partition of /dev/sda
/dev/loop0	B	First loopback device
/dev/console	C	System console
/dev/parport0	C	first parallel port
/dev/lp0	C	first printer port (LPT1)
/dev/null	C	System Trash Can (Black Hole)
/dev/tty$N$	C	N^{th} virtual console
/dev/pts/N	C	N^{th} pseudo terminal (dynamically created)
/dev/ttySN	C	N^{th} serial port
/dev/psaux	C	PS/2-style mouse port
/dev/mouse	L	Soft link to a serial mouse like psaux
/dev/printer	S	lpd local socket

Table 4.2: `Linux` Devices (See 4.1 for type info)

`/dev/null` is a funny device that swallows everything you feed it like a black hole, so don't put anything there that you want back. It's very convenient for swallowing up unwanted error messages like this:

```
<bash>: netscape >& /dev/null &
```

We will learn the meaning of the preceding example in the next section. The `cat` command prints a text file to the console and the `cp` command copies one file to another. Try this example

```
<bash>: cat .cshrc
<bash>: cp .cshrc /dev/null
<bash>: cat /dev/null
```

The last command should produce nothing.

4.5.1 Exercise: Redirect the output of command `top` to file top.log

4.5.2 Exercise:

```
<bash>: ls -l /dev | grep ^c
<bash>: ls -l /dev | grep ^b
<bash>: ls -l /dev | grep ^l
<bash>: ls -l /dev/hda*
<bash>: ls -l /dev/sda*
```

4.5.3 Discussion

Chapter 5

Process Control

In this chapter we will learn how to control your processes both interactively and non-interactively. It is very important to maintain control of your system and manage unruly processes.

5.1 Creating Foreground and Background Processes

Process control refers to the ability to control processes both in the *foreground* and *background* of the shell. A *foreground* process is one that is running interactively in your shell, and include most processes we have discussed so far. You can interrupt a *foreground* process by typing `Ctrl-Z` pronounced "control z" (hold <Ctrl> and hit the *z* key). Now to put that process back into the *foreground*, you use the (bash) command `fg`.

A *background* process, by contrast, is one that does not interact with your shell directly. It keeps to itself and runs silently. To place a process into the background you use the builtin command `bg`. From the shell point of view, all processes have a *job* number. From the system's point of view, all processes have a *process ID*, often called a *PID* (see the `ps` command).

To actually be able to catch one of these processes, we need to find a command that lasts longer than an instant (like `date`). Lets work with the `xterm` command which creates an X-terminal window:

```
<bash>: xterm              # Start a fresh xterm
Ctrl-Z                        # You hit Ctrl-Z and suspended it!
[1]+ Stopped   xterm

<bash>: fg                 # Restarted it with fg
xterm
Ctrl-Z                        # You suspended it again.. good..
[1]+ Stopped   xterm
```

```
<bash>: jobs              # Find the job number.
[3]+ Stopped  xterm
<bash>:

<bash>: bg %3            # Put xterm in the background.
[3]    xterm &
<bash>:
```

You should now have an `xterm` which behaves normally. The last [3] displayed indicates that its *Job Number* is 3. You can use that number to refer to it using `fg` as in

```
<bash>: fg %3                # foreground it with fg
Ctrl-Z                         # Suspend it again.
[3]+ Stopped  xterm

<bash>: bg %3                 # Put it in the background with bg
[3]     xterm &
<bash>: jobs
[1]     Running                         xterm -cr red -ms >&
[2]     Running                         xclock
[3]     Running                         xterm
<bash>: kill -9 %3
<bash>: jobs
[1]     Running                         xterm -cr red -ms >&
[2]     Running                         xclock
[3]-    Terminated                      xterm
```

The `jobs` displays all jobs in the shell along with their number and status. The `kill -9 %N` kills job number *N*. The flag (-9) is needed to force a "sure kill" which is sometimes needed. The ampersand symbol (&) is what you would use to put a command directly into the background with `xterm &`.

Be careful because some commands want to write errors to the standard output *stdout* and that could cause strange results on your screen. To avoid this, you can redirect *stdout* to `/dev/null` (or a log file) AND put it into the background in one gulp:

```
<bash>: xterm >& /dev/null &
or
<bash>: xterm >& /tmp/xterm-log &
```

This is advisable if you plan to script your commands, because *stdout* that has no place to go can stop your script in its tracks. Experience has shown to always deal with the trash in advance.

5.2 Killing Processes With `kill`

Now we will display the process numbers with the `ps` command

```
<bash>: ps
  PID TTY          TIME CMD
  683 ttyp1    00:00:00 bash
31666 ttyp1    00:00:00 xterm
32716 ttyp1    00:00:00 xclock
<bash>: ps -agl                # Get a long list with ps
  F   UID   PID  PPID PRI  NI   VSZ  RSS WCHAN  STAT TTY         TIME COMMAND
000   501   683   672  13   0  2220  804 rt_sig S    ttyp1      0:01 [bash]
000   501 31666   683   0   0  3012 1900 wait4  S    ttyp1      0:00 xterm -cr r
000   501 32716   683   0   0  2572 1304 do_sel S    ttyp1      0:00 xclock

<bash>: top
42 processes: 40 sleeping, 2 running, 0 zombie, 0 stopped
CPU states:  2.3% user,  0.7% system,  0.0% nice, 96.8% idle
Mem:   63484K av,  52328K used,  11156K free,  18020K shrd,   2896K buff
Swap: 220316K av,   4964K used, 215352K free                 28804K cached

  PID USER     PRI  NI  SIZE  RSS SHARE STAT  LIB %CPU %MEM   TIME COMMAND
  664 root      18   0 14824  14M  1540 R       0  1.1 22.7  18:21 X
31797 carinhas   9   0  7060 7060  3276 S       0  0.0 11.1   0:16 xemacs
  464 carinhas   0   0  6020 6020  3596 S       0  0.0  9.4   0:03 netscapex
31666 carinhas   0   0  1900 1900  1440 S       0  0.0  2.9   0:00 xterm
32716 carinhas   0   0  1304 1304  1096 S       0  0.0  2.0   0:00 xclock
... <Shift-M> Sorted by memory usage....
Ctrl-C <quit>
```

Now I have located a renegade process called `xclock` which seems using way too much memory according to the `top` utility. For example, if you want to kill process 32716, do the following:

```
<bash>: kill -9 32716
[3]    Terminated                      xclock
<bash>:
```

Study also the man pages for `killall`, `pstree`, and *proc* for the LPI exams.

Next is an abbreviated manual page for `kill`, but beware that most shells have their own built-in version of `kill`.

```
KILL(1)           Linux Programmer's Manual           KILL(1)
NAME
       kill - terminate a process
SYNOPSIS
       kill [ -s signal ] %job |  pid ...
       kill  -l [ signal ]
OPTIONS
       pid .. Specify the list of processes.
       -s     Specify  the  signal name or number to  send.
       -l     Print a list of signal names. See signal(7)
```

Here is the relevant and abbreviated man section of signal(7):

```
SIGNAL(7)         Linux Programmer's Manual          SIGNAL(7)
NAME      signal - list of available signals

      Signal       Value     Action   Comment
       ----------------------------------------------------------
       SIGHUP        1        A      Hangup detected on controlling terminal
       SIGINT        2        A      Interrupt from keyboard
       SIGQUIT       3        A      Quit from keyboard
       SIGILL        4        A      Illegal Instruction
       SIGABRT       6        C      Abort signal from abort(3)
       SIGFPE        8        C      Floating point exception
       SIGKILL       9       AEF     Kill signal
       SIGSEGV      11        C      Invalid memory reference
       SIGPIPE      13        A      Broken pipe: write to pipe with no readers
       SIGALRM      14        A      Timer signal from alarm(2)
       SIGTERM      15        A      Termination signal
       SIGUSR1   30,10,16     A      User-defined signal 1
       SIGUSR2   31,12,17     A      User-defined signal 2
       SIGCHLD   20,17,18     B      Child stopped or terminated
       SIGCONT   19,18,25            Continue if stopped
       SIGSTOP   17,19,23    DEF     Stop process
       SIGTSTP   18,20,24     D      Stop typed at tty
       SIGTTIN   21,21,26     D      tty input for background process
       SIGTTOU   22,22,27     D      tty output for background process

       The letters in the "Action" column have the following meanings:

       A      Default  action is to terminate the process.
       B      Default action is to ignore the signal.
       C      Default action is to dump core.
       D      Default action is to stop the process.
       E      Signal cannot be caught.
       F      Signal cannot be ignored.
       G      Not a POSIX.1 conformant signal.
```

5.3 Managing Process Priorities

In Linux and UNIX , all processes have an importance factor called a *priority*. The *priority* tells the system *kernel* how often to service that process. Obviously some processes are much more critical to the system than others. Hardware access is more important than email. The smaller the priority value, the more importance the process is given. Priority values range from -20 to 20, where -20 is the highest priority and 20 is the least.

You can see the priority of your processes by reading the *PRI* column in the `ps -xl` command:

```
<bash>: ps -xl

  F   UID   PID  PPID PRI  NI   VSZ  RSS WCHAN  STAT TTY        TIME COMMAND
000   501   617   612   0   0  2032  504 do_sel S    ?          0:03 fvwm
000   501   622   617   0   0  3144  872 do_sel S    ?          0:01 xterm -cr r
....etc....
```

There are two major tools for manipulating process priorities, `nice` and `renice`. `nice` lets you specify the priority for a command initially, and `renice` allows you to change the priority after the process is started. Here's an exercise for you:

```
<bash>: man nice
----------------------------------------------------------------
NICE(1)                                                   NICE(1)
       nice - run a program with modified scheduling priority
SYNOPSIS
       nice  -n priority command [arguments]
----------------------------------------------------------------
```

5.3.1 Exercises

```
<bash>: xclock -rv &
<bash>: nice -10 xclock -rv &
<bash>: /bin/nice -n 10 xclock -rv &          # Same as above
<bash>: ps -xl

  F   UID   PID  PPID PRI  NI   VSZ  RSS WCHAN  STAT TTY        TIME COMMAND
000   501   617   612   0   0  2032  504 do_sel S    ?          0:03 fvwm
000   501   622   617   0   0  3144  872 do_sel S    ?          0:01 xterm -cr r
000   501  2855  1321   8   4  2568 1300 do_sel SN   ttyp2      0:00 xclock

<bash>: renice 20 2855                      # Renice that xclock pid 2855.
2855: old priority 4, new priority 20       # Don't worry if numbers don't jive.

<bash>: renice 5 2855                       # Please renice to LOWER # ?
 renice: 2855: setpriority: Permission denied  # Only HIGHER #'s allowed!
```

<bash>: ps -xl

```
  F   UID   PID  PPID PRI  NI   VSZ  RSS WCHAN  STAT TTY        TIME COMMAND
000   501   617   612   0   0  2032  504 do_sel S    ?          0:03 fvwm
000   501   622   617   0   0  3144  872 do_sel S    ?          0:01 xterm -cr r
000   501  2937  1321  19  19  2568 1300 do_sel SN   ttyp2      0:00 xclock
```

Note that you can only renice to a higher number (thus lower priority) so the system gives an error message. Also, the listed *PRI* takes time to adjust to the *NI* (`nice`) value.

```
 LINUX WARNING! tcsh defines its OWN built-in version of nice:
```

```
TCSH(1)                                                     TCSH(1)
   Builtin commands
       nice  [+priority] [command]
```

5.4 Cron

Cron is a *daemon* that can automate tasks by periodically running them based on a time scheme. Its very useful for maintenance tasks. *Cron* takes a filename as a script for input that it incorporates internally and checks periodically. The file is called a *crontab* file.

5.4.1 Making a Crontab File

The *crontab* file is by default a *bash* style script. It has extra parameters that tell *cron* when to start execution. Here is the format:

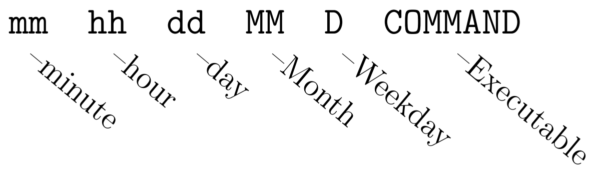


Figure 5.1: Crontab File Specification Format

Stars (*) are wildcards which always match, and rely on the other parameters to give *cron* timing queues. If you want to execute every day, just put a * in the day column.

A simple example of this would be to run a script above at 12 midnight every night:

```
# File: crontab-file. Execute me at 00.00 hours EVERYDAY!
  0 0 * * * /home/joe/bin/backup >& /home/joe/backup-log &
```

Notice that we have carefully routed *stdout* to a log file, otherwise *cron* could hang the job, or send us all the error by mail.

Another interesting property of the *cron* time spec is it's ability to specify numbers in a *modulo* fashion. Just divide a * by a number to get cron to execute a periodic command:

```
# File: crontab-file. Execute backups every four hours.
  * */4 * * * /home/joe/bin/backup >> /home/joe/backup-log 2>&1
```

5.4.2 Submitting Cron Jobs

Once we have written our *crontab* file, we can submit it by doing a `crontab crontab-file`. `crontab -l` lists the current crontab entries for you.

```
<bash>: crontab crontab-file
<bash>: crontab -l
# DO NOT EDIT THIS FILE - edit the master and reinstall.
# (crontabs installed on Mon Feb  7 00:13:36 2000)
# (Cron version -- $Id: crontab.c,v 2.13 1994/01/17 03:20:37 vixie Exp $)
# Execute the job every four hours.
# /home/joe/crontab-file
  * 0 * * * /home/joe/bin/backup >> /home/joe/backup-log 2>&1
```

You may also edit the current *crontab* file with the `crontab -e` command, which will throw you into your default editor. When you exit, the crontab file is automatically updated.

5.4.3 Removing Cron Jobs

You can always remove your `cron` jobs by using the (-r) flags:

```
<bash>: crontab -r
<bash>: crontab -l
```

5.4.4 Exercises

Exercise 1:

Create a simple script named `hello` that prints out "hello joe". It should only have two lines:

```
#!/bin/bash
echo Hello joe you slacker...
```

This file should be placed in your personal `~/bin` directory, and should have the appropriate file mode for execution (hint: chmod). Make sure to test this script by hand before you go on.

Exercise 2:

Now create a crontab file named `crontab-file` that will execute `hello` every 2 minutes. Submit this crontab file as discussed above and check your mail on the even minutes to see if the scripts work.

Note: Cron's default behavior is to send any unhandled output to your mail. If mail is not working for some reason, you may send your output to a file or to your screen.

Chapter 6

Text Editing Tools

6.1 Vi

Although `vi` is considered one of the most primitive editors in `Linux` , it is still the most native of the `UNIX` editors. In a pinch, you will have to know how to use `vi` if your favorite editor is not available.

`vi` has 2 basic modes: Command mode, and Insert mode. First we will start to work on our project. Basic command mode functions are listed below. Those with and star beside them indicate commands which enter insert mode.

key	Insert	Description
A	*	Append after this line
a	*	Append after this character
C	*	Replace to end of line
cw	*	Replace this word
i	*	Insert (start typing here)
O	*	Open to the line above cursor
o	*	Open to the line below cursor
R	*	Overwrite Mode
r		Overwrite with the next char
j		Move cursor one line downward
k		Move cursor one line up
l		Move cursor one char right
h		Move cursor one char left
x		Delete character
w		Move one word forward
b		Move one word back
D		Delete to end of line
dd		Delete this line
dw		Delete word
yy		Yank a line into buffer
p		Insert buffer lines after line
P		Insert buffer before this line
<esc>		Exit insert mode

Table 6.1: `vi` Command Mode Operations

These commands may not seem obvious from a mnemonic point of view, but were designed for simplicity and keyboard efficiency.

You can read and write files in command mode by using `ex` mode which is invoked by typing : (which will prompt you at the bottom of the `vi` screen)

```
:w        Write this file out.
:q        Quit vi w/o writing.
:q!       Force quit vi w/o writing.
:w!       Force write.
:wq       Write this file AND quit vi.
:r file   Read in ``file'' before cursor.
```

Find, replace, and execute commands are done in a similar way:

```
/pattern                Find ``pattern'' and move cursor there
:1,100 s/blue/green/g   For line 1 to 100, substitute green for blue
:1,$   s/blue/green/g   For all lines, substitute green for blue
:'a,'b s/pie/cake/g     From mark ``a'' to mark ``b'', substitute out pie
:1,$   !sort            sort all lines
```

Once we get familiar with `vi` we will discuss these modes further. The first thing to do is to start the editor by typing the following:

```
<bash>: vi input.txt
```

You should now have an empty screen that looks like

```
█
~
~
~
```

Now type `i` and start typing some meaningful text like

```
Mary had a little lambda, who's feet was white as snow.
~<esc>
~
~
```

The `i` puts you into insert mode, and the final `<esc>` puts you back into command mode. Now write and quit by doing a `:wq`

```
Mary had a little lambda, whose fleets was white as snow.
~
~
~
~
:wq
"input.txt" 1 line, 58 characters written
```

6.2 Editing With `Pico`

`Pico` is by far the easiest editor to use since it usually displays its commands at the bottom of the screen. Start by typing `pico input.txt`:

```
   UW PICO(tm) 3.5                    File: input.txt
Mary had a little foo, whose fleece was white as snow....
Start xxx Regular
   bar baz foo them@nowhere.com Regular
Linux Rocks TurboLinux eats apples Respect
Some say Linux Rocks

^G Get Help  ^O WriteOut  ^R Read File  ^Y Prev Pg  ^K Cut Text  ^C Cur Pos
^X Exit      ^J Justify   ^W Where is   ^V Next Pg  ^U UnCut     ^T To Spell
```

The only problem with `pico` (and other non-vi editors) is that it may not be on your system in an emergency situation, which is when you need it the most. Hit `^X` to exit `pico`.

6.3 Editing With `emacs`

`emacs` is the most powerful off all the editors commonly used under `Linux` . The *X-Windows* version is fully loaded with all kinds of option menus and panels. Despite its complexity, you can get by with a handful of commands. In this section, C- represents Ctrl-, so for example C-x is Ctrl-x.

emacs Exiting and File Manipulation

save a file back to disk	**C-x C-s**
save **all** files	**C-x s**
exit emacs permanently	**C-x C-c**
read a file into emacs	**C-x C-f**
insert contents of another file into this buffer	**C-x i**
suspend emacs (or iconify it under X)	**C-z**
goto beginning of line	**C-a**
goto end of line	**C-e**
delete character	**C-d**
cut to end of line	**C-k**
cut and buffer text	**C-w**
set mark	**C-space**
paste buffer	**C-y**

emacs Error Recovery and Searching

undo an unwanted change	**C-x u**
abort partially typed or executing command	**C-g**
redraw garbaged screen	**C-l**
search forward	**C-s**
search backward	**C-r**
regular expression search	**C-M-s**
reverse regular expression search	**C-M-r**

emacs Formatting

indent current **line** (mode-dependent)	**TAB**
indent **region** (mode-dependent)	**C-M-**
indent **sexp** (mode-dependent)	**C-M-q**
indent region rigidly *arg* columns	**C-x TAB**

emacs Help

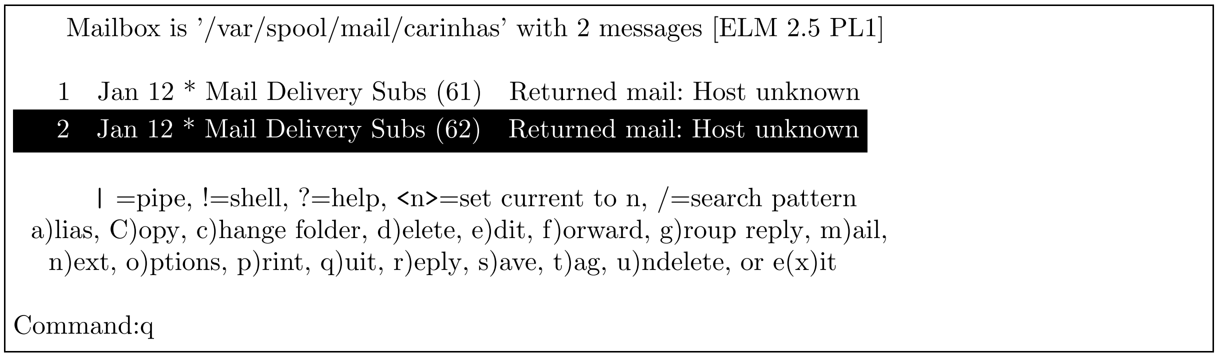
basic help	**C-h**
scroll help window	**C-M-v**
emacs Tutorial	**C-h t**
apropos: show commands matching a string	**C-h a**
show the function a key runs	**C-h c**
describe a function	**C-h f**

6.4 Using Mail

Linux comes with several mail handlers. I prefer elm because I know it well and it is powerful, however pine is very user friendly and easy to use. We will discuss both.

6.4.1 Elm

Fire up elm by typing "elm". When it first starts, it will ask you some questions about files, just say yes to all.

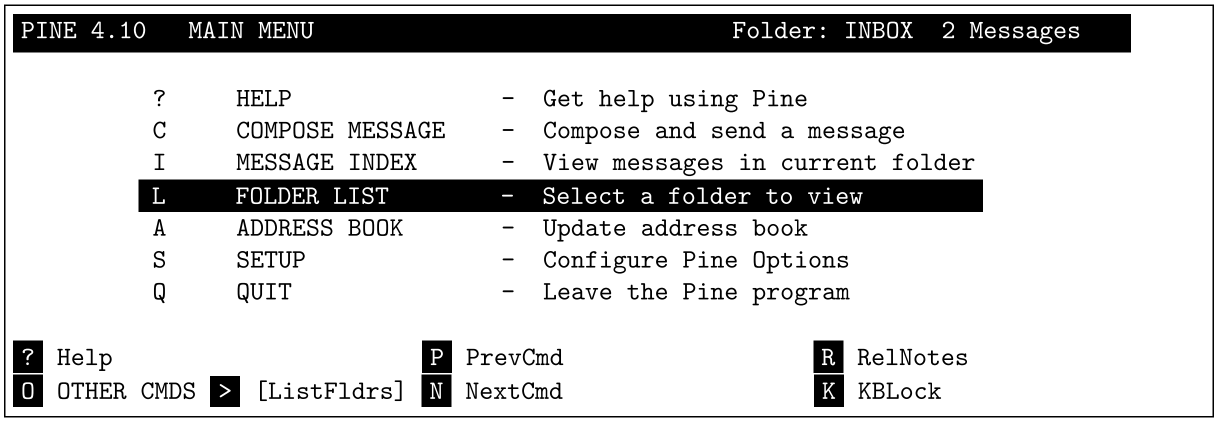


```
      Mailbox is '/var/spool/mail/carinhas' with 2 messages [ELM 2.5 PL1]

   1   Jan 12 * Mail Delivery Subs (61)   Returned mail: Host unknown
   2   Jan 12 * Mail Delivery Subs (62)   Returned mail: Host unknown

         | =pipe, !=shell, ?=help, <n>=set current to n, /=search pattern
  a)lias, C)opy, c)hange folder, d)elete, e)dit, f)orward, g)roup reply, m)ail,
    n)ext, o)ptions, p)rint, q)uit, r)eply, s)ave, t)ag, u)ndelete, or e(x)it

Command:q
```

Once inside elm, just hit "m" at the command prompt, and follow the signs. elm is more command driven and UNIX -like. When in doubt, just keep hitting "q" for quit. Use the arrow keys to go up and down the message list.

6.4.2 Pine

Pine is easier to learn and use, and is compatible with elm, so you can use them interchangeably. Use the up-arrow, down-arrow, <, and > arrows to navigate.



```
PINE 4.10   MAIN MENU                               Folder: INBOX  2 Messages

          ?     HELP               -  Get help using Pine
          C     COMPOSE MESSAGE    -  Compose and send a message
          I     MESSAGE INDEX      -  View messages in current folder
          L     FOLDER LIST        -  Select a folder to view
          A     ADDRESS BOOK       -  Update address book
          S     SETUP              -  Configure Pine Options
          Q     QUIT               -  Leave the Pine program

? Help                       P PrevCmd                  R RelNotes
O OTHER CMDS > [ListFldrs]   N NextCmd                  K KBLock
```

6.5 Regular Expressions

Regular Expressions form one of the pillars of `UNIX` obscurity in text processing. Regular Expressions are search patterns that give you ominous control over how text patterns are found. Here are the basics pattern matching symbols:

`.`	Match anything. ‘‘.*’’ matches anything any number of times. See *
`*`	Match the last char or exp 0 or more times. a* matches (), a, aa, aaa,
`+`	Match the last char or exp 1 or more times. a+ matches a, aa, aaa, ..
`?`	Match the last char or exp 0 or 1 time. a? matches (), a
`^`	Match the beginning of the line. ^abc matches lines starting with abc.
`$`	Match the end of a line. ^$ Matches empty lines.
`[]`	Match chars inside. [abc] can match a, b, or c.
`[^]`	Negates the match chars: [^abc] matches anything except a, b, or c.
`()`	Char grouping. (abc)+ matches abc, abcabc, abcabcabc, ...
`\`	Match regex chars like \+, *, and \^.
{n}	The preceding item is matched exactly n times.
{n,}	The preceding item is matched n or more times.
{,m}	The preceding item is optional and is matched at most m times.
{n,m}	The preceding is matched at least n times, not more than m times.
`\b`	Match a word boundary: abc\b matches **abc** but not abcd.
`\B`	Match a non-boundary: abc\B matches **abc**d but not abc.
`\w`	Match a word: abc\w matches **abcdef** but not abc@.
`\W`	Match a non-word: abc\W matches **abc&** but not abc.

6.5.1 GREP

`grep` is a pattern search tool. It will find patterns in a file and print it on the terminal. These patterns are *regular expressions* as well. Test `grep` with the file as before.

`grep` *[options] pattern file..*

```
# 1. Find lines with ‘‘Linux’’ ANYWHERE in the file.
grep Linux input.txt

# 2. Find ‘‘Linux’’ in a non word boundary like ‘‘TurboLinux’’
grep ‘‘\BLinux’’ input.txt    # Remember \B?

# 3. Find ‘‘Linux’’ at beginning and the end of line, regular expressions.
grep ’Linux$’ input.txt       # Note the single quotes to trick the shell,
grep ^Linux input.txt          # because shell tries to interpret stuff like $.

# 4. Lines start with ‘‘Linux’’, end with ‘‘Respect’’.
grep ’^Linux.*Respect$’ input.txt  #  ‘‘.*’’ means any number of anything.

# 5. Find some tough patterns:
 grep ’<.*>’ input.txt       # Find lines that have ‘‘<something>’’. Html tags?
```

6.5.2 AWK

`Awk` is a pattern scanning and process language. Its Syntax is

```
awk 'program' input-files
```

When the program is long, it is usually more convenient to put it in a file and run it with a command like this:

```
awk -f program-file input-files
```

Take a look at the manual now. Here are some one-liner examples:

```
awk '/xxx/' input.txt  # Find ''xxx'' in input.txt
    Most all regular expression syntax works inside the //'s.
awk 'NF > 0' input.txt
    This program prints every line that has at least one field. Its an
    easy way to delete blank lines from a file.
awk -F: '{ print $1 }' /etc/passwd | sort
    This program prints a sorted list of the login names of all users.
awk 'END { print NR }' input.txt
    This program counts lines in input.txt.
awk 'NR % 2 == 0' input.txt
    This program prints the even numbered lines of input.txt.
    If you'd use the expression 'NR % 2 == 1', it'd print odd numbered lines.
awk '{if (length($0) > max) max=length($0)} END {print max}' input.tx
    This program prints the length of the longest input line.
awk 'length($0) > 80' input.txt # Print lines longer than 80 characters.
    Since the sole rule has a relational expression as its pattern,
    and has no action, the default action, printing the record, is used.
awk 'BEGIN { for (i = 1; i <= 7; i++) print int(101 * rand()) }'
    This program prints seven random numbers from zero to 100, inclusive.
ls -lg *.txt | awk '{ x += $5 } ; END {print "total bytes:" x }'
    This program prints the total number of bytes used by all .txt files.
```

6.5.3 Perl

`Perl`, "Practical Extraction and Report Language" is the Swiss Army Knife of all scripting languages. It has everything except a Megawatt laser, and a kitchen sink. Perl has a very complete set of *regular expressions* and a large set of `UNIX` system calls that make it a steroid-pumped tool:

`perl` *`[options] [ program ]`*

On the command line you can use the syntax

`perl` *`-ne 'command' file`*

where the options are

-e 'command'	**execute *command* instead of script-file**
-n	**run the script on each line of input *file***
-w	**Print WARNINGS for your script *file***

Even though complex, we can learn a few `perl` one liners that work miracles:

```
# 1. Find lines starting with ``Linux Rocks'' in input.txt.
perl -ne 'if ( /^Linux Rocks/ ) {print} ' input.txt

# 2. Find ``Linux'' in a non word boundary like ``TurboLinux''
perl -ne 'if ( /\BLinux/ ) {print} ' input.txt

# 3. Find ``Linux Rocks'' and replace it with ``Rocks Linux''.
#    Regular expressions matching ()'s get named $1, $2, $3, etc..
perl -ne 's/^(Linux) (Rocks)/$2 $1/;  print ' input.txt

# 4. Lines start with ``Linux'', end with ``Rocks''.
perl -ne 'print if /^Linux Rocks$/ ' input.txt

# 5. Find email addresses in input.txt:
perl -ne 'print if (/(\w[.\w]*\@\w[\w_-]+\w)/gm)' input.txt
```

6.5.4 SED

sed is a stream editor, which means that it works like a conveyer belt on your terminal. The text flies past you while it works.

sed *[script] file..*

sed is complex so we'll just show some simple usage and examples:

```
# 1. From line 1 to 32, print the line number
sed '1,32 {= ; }' input.txt   # Almost works in VI

# 2. Delete lines 1 through 32
sed "1,32 d" input.txt        # :1,32 d in VI

# 3. Under UNIX: convert DOS newlines (CR/LF) to Unix format
sed 's/^M$//' input.txt       # In bash ^M = C-v C-m, tcsh ^M = C-v C-j

# 4. Under DOS: convert Unix newlines (LF) to DOS format
sed 's/$//' input.txt         # method 1
sed -n p input.txt            # method 2

# 5. Delete leading whitespace (spaces/tabs) from front of each line
#    '^t' represents a true tab character. ^t => Ctrl-V Ctrl-I.
sed 's/^[ ^t]*//' input.txt   # OK in VI without single quotes.

# 6. Delete trailing whitespace (spaces/tabs) from end of each line
sed 's/[ ^t]*$//' input.txt               # see note on '^t', above

# 7. Delete BOTH leading and trailing whitespace from each line
sed 's/^[ ^t]*//;s/[ ^t]*$//' input.txt    # see note on '^t', above

# 8. Substitute "foo" with "bar"
sed 's/foo/bar/' input.txt       # replaces 1st instance foo with bar.
sed 's/foo/bar/4' input.txt      # replaces only 4th instance in a line
sed 's/foo/bar/g' input.txt      # replaces ALL instances within a line
sed '/baz/s/foo/bar/g' input.txt  # Sub ONLY on lines which contain "baz"

# 9. Match lines that have ``Regular'' in them:
sed '/^Regular/d' input.txt      # Delete lines starting with ``Regular''
sed '/Regular$/d' input.txt      # Delete lines ending with ``Regular''

#10. From ``Start'' to ``Linux'', substitute apples for oranges
sed  /Start/,/Linux/s/apples/oranges/ input.txt
```

Chapter 7

Shell Scripting

There's nothing more aggravating than having to type out long commands over and over again. Shell scripts are simple text files that list shell commands into a convenient "tool" which you can use more easily than typing in all those commands over and over.

Now that we know enough shell workings to be dangerous, we need to focus our attention to making changes permanent. It is essential to have easy to use environments and scripts to do complex tasks, because otherwise we are stuck repeating numerous small commands daily. The natural starting point is to customize our `.bashrc` and `.cshrc` files with a few aliases and a prompt or two.

7.1 Shell Initialization Files

We're going to develop simple aliases and shell variables for our bash shell that will make life easier when you log into your Linux system.

7.1.1 Initialization Files for Bash

Start editing `.bashrc` now. Write in the following aliases and prompt definitions:

```
alias m='less -EMsX '
alias h='history   | tail -40'
alias d='ls -l $*  | less -EMsX'
alias dt='ls -blt  $* |less -EMsX'
alias Dt='ls -aglt $* |less -EMsX'
PS1='[\s]:\u: '
```

The *$** represents all the remaining variables on the command line except for the calling command. It is equivalent to all subsequent command line variables, *"$1 $2 ..."*.

After you modify `.bashrc` you must do a "`source .bashrc`" to initialize your changes. Type `alias` to see what the system's definitions, and try out our new aliases.

Remember we can also have a .bash_profile which is supposed to carry your environmental variables:

```
# .bash_profile
# Get the aliases and functions
if [ -f ~/.bashrc ]; then
        . ~/.bashrc
fi

# User specific environment and startup programs
PATH=$PATH:/usr/local/bin:$HOME/bin
ENV=$HOME/.bashrc
USERNAME=""

export USERNAME ENV PATH
```

Note the final export at the end of the file.

7.1.2 Initialization Files for Tcsh

Now lets make the analogous aliases in your .cshrc:

```
alias lester "less -EsX"              # Use less as our pager..
alias h    'history \!* | tail -40 | lester'
alias m     lester                    # Used to be more, now more is less.
alias d     'ls -l    \!* | lester'
alias dt    'ls -lt   \!* | lester' # Sort by chronological order.
alias Dt    'ls -laFt \!* | lester' # Same as above, show dot (.xxx) files.
set prompt="%n:%~: "                  # "user:/location/location/location: "
```

Try these out in tcsh after executing

<tcsh>: source .cshrc

Now submit these changes by typing `source .cshrc`. Type `alias` to see what the system's definitions, and try some of them out. Both `Tcsh` and `Csh` use .cshrc and .login. The .login file is normally used to store environment variables and *path* information, while .cshrc normally hold aliases and prompt information.

7.2 Utility Shell Scripts

Our first goal is to create a script utility that will print files in a directory called `/home/joe/address`. The syntax will be

`phone.shell` *`file1 file2 ....`*

Because of `tcsh's` simple and clear syntax, we will first use the `tcsh` language to construct a solution. Afterwards, we show the equivalent `bash` solution.

7.2.1 Tcsh Solution

The first thing we want to do is to tell the script that it is a `csh` script, and what the acceptable number of parameters are. In the following scripts make sure you ALWAYS remove the comment on the first line, otherwise these examples will not function. In general, students should NOT write any comments in these examples.

Start by editing a file named `~/bin/phone.csh`:

```
#! /bin/csh -f   # This is not a comment. Remove from 2nd #.
#
set a=$0         # $0 refers to the name of this script.
if($#argv < 1 ) then
  echo Usage: $a:t 'Name'
  exit(1)
endif
```

The term *set a=$0* refers to the 0th component of the command line, which is the command file, `phone.csh`, in our case. The *$a:t* instructs `csh` to remove all the leading pathname components, since we are uninterested in seeing the full path. Test this by invoking the name of the script:

```
<bash>: phone.csh
Usage: phone.csh Name
```

Did you remember to `chmod` our script so that it is executable?

So far the script does nothing but tell us if we have the right format. Next we will add a *foreach* line that will give us control for each filename:

```
foreach name ($*)
  echo ~/address/$name
end
```

Now your script should print each filename you enter on the command-line, as a check in our progress. Again, test this script and check that the loop works:

```
<bash>: phone.csh red green blue

/home/joe/address/red
/home/joe/address/green
/home/joe/address/blue
```

Now that we know it works, replace the `echo` command with the correct `cat` phrase. While we are at it, we will check for the existence of each file with the (-e filename) test. The previous *foreach* clause should **NOW** look like:

```
foreach name ($*)
  if (-e ~/address/$name) then        # The (-e) flag checks for existence.
    cat ~/address/$name
  else
    echo "no such file $name"
  endif
  echo "-------------------------"    # Separate entries with a line.
end
```

The final script should resemble the following:

```
#! /bin/csh -f                     # This a csh script, omit this comment!
# phone.csh                        # Comment to give the filename.
set a=$0                           # $0 refers to the name of this script.
if($#argv < 1 ) then               # If right number of arguements,
  echo Usage: $a:t 'name'          #   Echo the proper syntax
  exit(1)                          #   Exit with error code 1
endif                              # End if
#
foreach name ($*)                  # For each name on the command line.
  if (-e ~/address/$name) then     #   If file exists (-e),
    cat ~/address/$name            #     List the file
  else                             #   Else
   echo "no such file $name"       #     Barf
  endif                            #   End if
  echo "---------------------"     # Separate entries with a line, cosmetic.
end                                # Bye-Bye
```

Lets check our program by creating an ~/address directory and several files there...

```
<bash>: mkdir ~/address
<bash>: cd ~/address
<bash>: echo hello1 says 123 > hello1      # Creates a file called hello1.
<bash>: echo hello2 says 456 > hello2      # Creates a file called hello2.
<bash>: echo hello3 says 789 > hello3      # Creates a file called hello3.
<bash>: phone.csh hello1 hello2 hello3

hello1 says 123
------------------------
hello2 says 456
------------------------
hello3 says 789
------------------------
```

7.2.2 Bash Solution

The `bash` solution to the exercise is not as obvious as the `csh` one because of `bash's` bizarre syntax. Call this file phone.bash:

```
#! /bin/bash                              # This is a bash script. Omit comment.
if [ "$#" -lt 1 ] ; then                  # If number of arguments too small
   echo Usage: $0 args                    #    State the correct usage
   exit 1                                 #    Exit with error status
fi                                        # Note "fi" closes "if" statements
for i in $@ ; do                          # Start a "do" loop on the $@ array
   if [ -f ~/address/$i ];then            #    If the file exists.
      cat ~/address/$i  ;                 #       print it,
   else                                   #    Else
     echo There is no file $i             #      Error message if absent.
   fi                                     #    End if
echo --------------                       # Print a separator between files
done                                      # "done" closes the "do" loop
```

7.2.3 Exercises

Exercise 1: Make the phone.bash version above work.

Exercise 2: Add an extra error message line if the file does not exist.

7.3 General Scripting

In this section we will create a several script programs named “doubletake” that will remove repeat entries in a list. We will use several of our newly learned languages to accomplish that task.

7.3.1 `awk` version

Edit doubletake and put in the following lines:

```
#! /bin/csh -f
# This script remove double in a sorted file. See the uniq manual.
set file1 = $1         # $1 is going to be the filename.
cat $file1 | awk '{ if ($0 != last ) {print $0; }; }{ last = $0; }'
```

The script first stores the first command argument in *$1'*. It then `cat`'s the file into `awk`. Notice you need the initial line of

#! /bin/csh -f

to tell the system that this is indeed a `csh` script.

Now the script that `awk` uses does 2 things. It first checks to see if the variable *last* was seen, if not, prints it. The second thing it does is to store the current line in *last*. Of course you could just type the whole second line into the command line, but that's not much fun if you have to do it 9 times a day.

7.3.2 Perl Version

```
#! /usr/bin/perl -w
my $line = "";                # declare this variable.
foreach $filename (@ARGV)     # cycle through each filename in the ARGV list.
{
   open (SF, $filename) or die "Can't open $filename" ;  # open the file and complain if not.
   while (<SF>)               # For every line in the file
   {
      chomp($_);              # Remove tailing whitespace and/or linefeeds.
      if ( $line ne $_ )      # ne = not equal
      {
         print "$line \n";    # Yes, print me.
      }
      $line = $_ ;            # Reset $line.
   }
}
```

7.4 Script Automation

Shell scripts are great tools when you can use them, and can save a lot of time. If you need to run applications or backups at odd hours or nightly, you really want to automate the task by using the `cron` tools, which we have already covered in chapter 5.4.

The following exercise describes how to set up a script program to do your dirty work while you are playing golf or sleeping.

7.4.1 Getting Scripts in Order

Before you submit a job for `cron` to handle, you must make sure you can run the job by hand. This involves making every command self sufficient by providing complete paths to the commands and setting the right permissions. Lets take backups for example. You need to have privileges to write to the tape device as a normal user unless you plan to run your jobs as **root** .

Here is a typical script, `/home/guest/bin/backup.sh` that would back up your home directory to the floppy device `/dev/fd0`. In this example, the floppy is being used as a character device just like a tape drive would be used:

```
#! /bin/bash
# This file is named backup.sh
cd /home/guest                              # Make sure you are at home.
/bin/tar cvfz /mnt/fd0 .                    # Back up your home directory
echo All done boss!
```

In this example, we have specified all paths to commands. This is required since your normal path/shell is not invoked in a script like this one. This script will create a lot of output which must be sent into a log file or `/dev/null` later when we make our crontab file.

7.4.2 Testing the Scripts

You must make sure you can run the scripts by hand before you try to invoke `cron`. This is typical for all programs that must be automated. Also make sure as mentioned before that all commands are given by their full path names. BTW, why doesn't the `echo` command need a path?

7.4.3 Exercises

Ex 1: Once you are confident that your script will run on its own two feet, you are ready to set it up with `cron`. Create a crontab file `/home/guest/crontab.backup` and submit it with

```
<bash>: crontab crontab.backup
```

as done in section 5.4. Ask for help if you get stuck.

Chapter 8

Using X-Windows

The X-Windows system is the `Linux` answer to a *graphical user interface* or GUI. It allows users a great interface to work from. To start X from the console, just type

```
startx
```

If everything works, you should have a terminal that has a few big black windows and a clock or two. This is just a single version of how X can be presented. X has many toys to play with. Here are a few:

Basic X Tools

xterm *[options]*	Start an X-Terminal
xclock *[options]*	A basic wall clock
xman *-options*	Display the manual X-style
xbiff *-option*	Mailbox flag to keep you waiting for mail
xboard	A chess game
xrdb *-merge Xfile*	Incorporate *Xfile* into your environment
xload	Show the system's usage load

There are actually too many applications in X to list. Almost everything these days has an X interface.

8.1 Customizing the X Environment

Just about everything in X can be customized, depending on how hard you want to work on it. The most simple thing you can do to customize your environment is to keep a defaults file, normally called `.Xdefaults` located in your home directory.

A typical .Xdefaults looks like this:

```
!---------------------------------
! .Xdefaults X-Resource File
!---------------------------------
Netscape.Navigator.geometry: =700x700
XTerm*visualBell  : on
XTerm*Font        : 9x15bold
XTerm*scrollBar   : on
XTerm*saveLines   : 4940
XTerm*borderColor : white
xclock*geometry   : 60x60-0+0
xclock*background : black
xclock*foreground : white
```

The initial instance in the variables is a *class* name. The subsequent variables in string are called *X-resources* or just *Resources* which identify the property of the program or property to modify. The *'s are wildcards and can match anything.

Finding what is and isn't a valid *X-resource* can be difficult, but normally most man pages have a listing of these and examples of how to set them. Once you have edited this file to your liking, you must activate the changes by using the xrdb utility like this:

```
<bash>: xrdb -merge .Xdefaults
```

Afterwards, try a command to see if your changes take hold. Normally, there is an initialization script in a system folder or in your home directory that will read your .Xdefaults for you when you start your X session. This file is sometimes called .xinitrc and is easy to customize:

```
#!/bin/sh
userresources=$home/.Xdefaults
usermodmap=$home/.Xmodmap
sysresources=/usr/X11R6/lib/X11/xinit/.Xresources

# merge in Xdefaults and keymaps
if [ -f $home/.netscape/lock ]; then
    rm -rf $home/.netscape/lock
fi
if [ -f $sysresources ]; then
    /usr/bin/X11/xrdb -merge $sysresources
fi
if [ -f $sysmodmap ]; then
    xmodmap $sysmodmap
fi

# start some nice programs
xterm &
xclock &
# fix the bell- 11 Nov 96 rcr
/usr/bin/X11/xset b 50 4500 100 &
exec fvwm
```

8.2 Window Managers

Unlike M$ GUI's, the X environment has many *managers* that completely control the *look and feel* of the environment. The one you are probably using now is called *fvwm* and is much simpler than the *Gnome* manager you are accustomed to using in *RedHat*. Here is a brief description of the managers (and some of their config files) that does no justice to any of them:

8.2.1 Gnome

Gnome is one of the two very popular X-window managers out there.

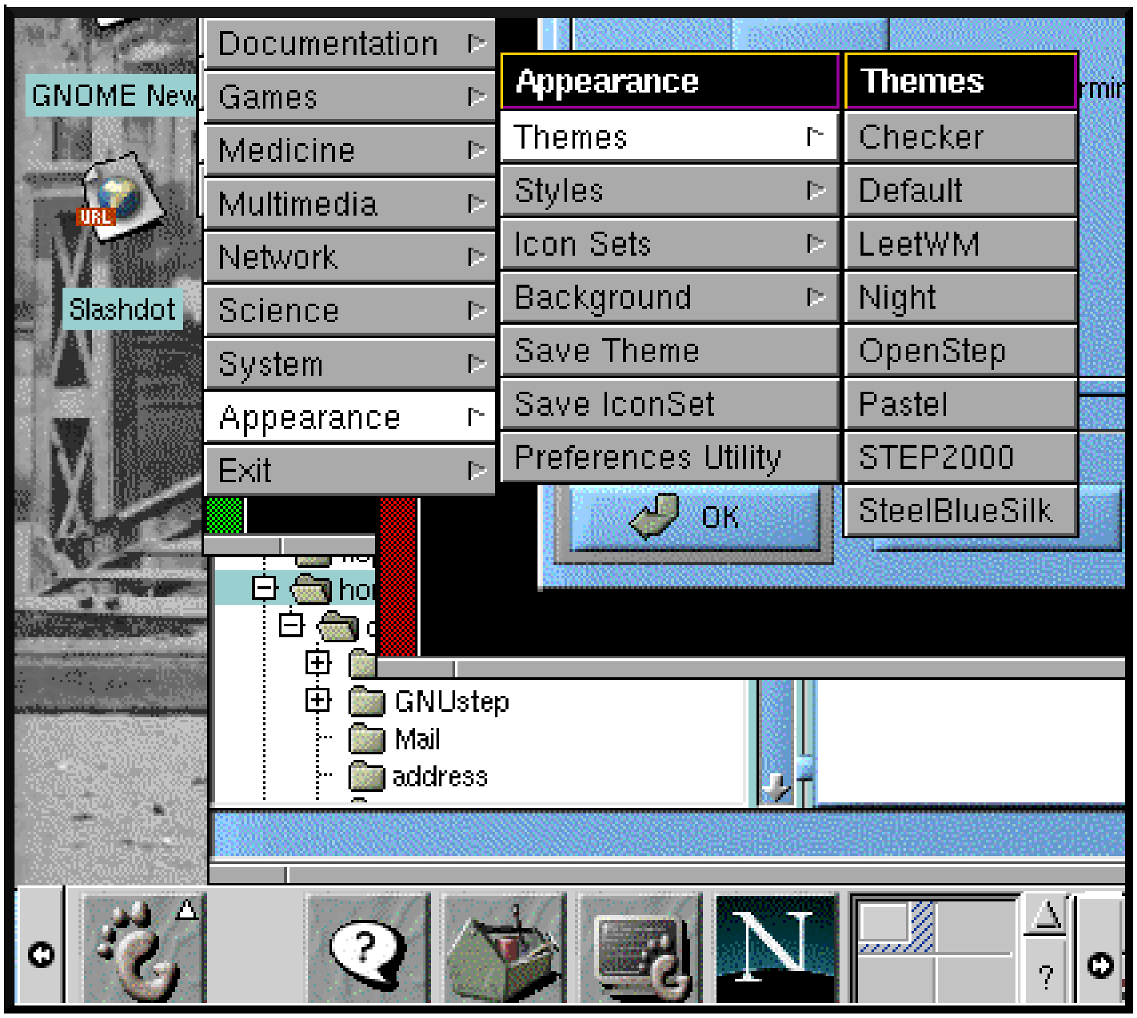


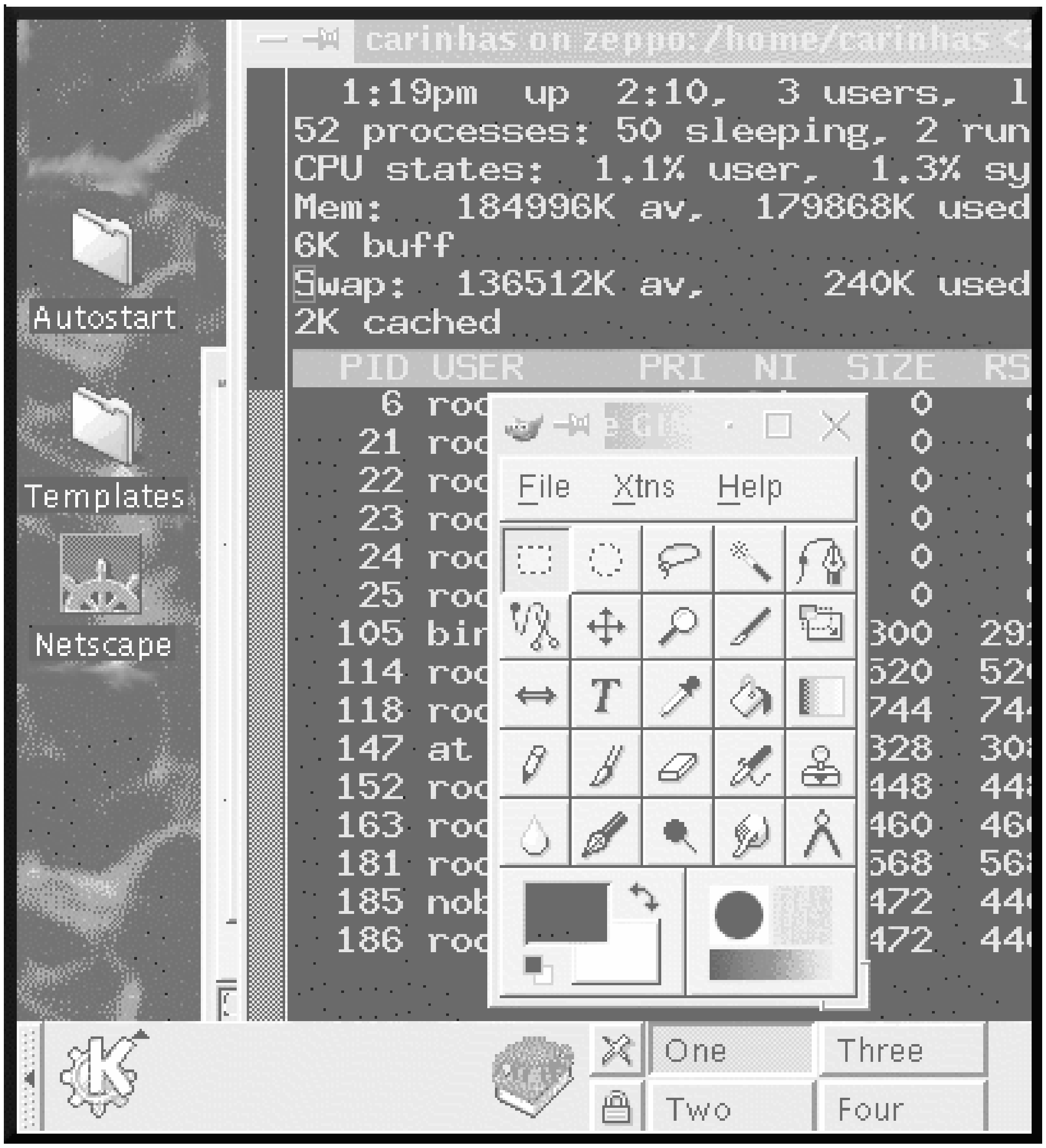
Figure 8.1: Gnome Interface

Lets look at Figure 8.1. First notice that the bottom panel has several typical icons like a tool-box, terminal window, and netscape.

It has a four panel virtual display which gives you four whole screens to work in. This gives some room to spread out. Inside the window we see some drop down panels that set the appearance. There is also the file manager visible. There are click-able icons on the desktop itself, much like the other OS's you have used. Not too shabby for a free OS..

8.2.2 KDE

KDE is the other popular X-window manager. Here is a sample of what the KDE looks like:



```
  1:19pm  up  2:10,  3 users,  l
52 processes: 50 sleeping, 2 run
CPU states:  1.1% user,  1.3% sy
Mem:   184996K av,  179868K used
6K buff
Swap:  136512K av,     240K used
2K cached
  PID USER      PRI  NI  SIZE  RS
    6 roc                   0
   21 roc                   0
   22 roc                   0
   23 roc                   0
   24 roc                   0
   25 roc                   0
  105 bir                 300  29
  114 roc                 520  52
  118 roc                 744  74
  147 at                  328  30
  152 roc                 448  44
  163 roc                 460  46
  181 roc                 568  56
  185 nob                 472  44
  186 roc                 472  44
```

Figure 8.2: KDE Interface

Notice that KDE also has four virtual panels to work on, and has all the bells and whistles a high quality GUI would provide.

You can get more info on Gnome and KDE at the X-User's Howto at `http://www.linuxdoc.org/HOWTO/`, the KDE homepage `http://www.kde.org`, and the Gnome homepage `http://www.gnome.org`.

Here is a list of the most popular window managers out there:

X-Window Managers for `Linux`

TWM *(.twmrc)*	One of the oldest and simplest managers. Very basic.
FVWM *(.fvwmrc)*	Young nephew of TWM
FVWM2 *(.fvwm2rc)*	Clever brother of FVWM
GNOME *(.gnome* dirs)*	An entire environment, more than just a manager.
KDE	Like GNOME, KDE is an environment, and uses the QT X-library.
AfterStep	NextStep feel and beyond
Window Maker	NextStep look and feel. Easy to configure
Enlightenment	An extreme, detailed, and configurable environment. Crashes

Chapter 9

TCP/IP Networking Basics

`Linux` comes with all the TCP/IP networking goodies you would expect from a full-featured `UNIX` box. It has `telnet`, `ftp`, `mail`, and many other services. Here's a quick list of the bread-n-butter commands:

Basic Network Applications and Services

telnet *RemoteHost*	Connect to another system (insecure)
rlogin *RemoteHost*	Login to RemoteHost (insecure)
ftp *RemoteHost*	*File Transfer Protocol* transfers files between hosts
ssh *RemoteHost*	Secure connection to RemoteHost
scp *file RemoteHost*	Secure copy *file* from here to RemoteHost
scp *Host1:f1 Host2:f2*	Secure copy *f1* from Host1 to Host2

TCP/IP Diagnostics

ping *RemoteHost*	Send a "ping" to a remote host to check connectivity
nslookup *RemoteHost*	Query a DNS server for hostname resolution
route	Print out the local routing table (check `/sbin/route`)
ifconfig	Print local TCP/IP network interface configuration
arp	print/manipulate system ARP cache
tcpdump *-i ethX*	listen to traffic on *ethX*

Networking Terms

TCP/IP	Transmission Control Protocol / Internet Protocol
ARP	Address Resolution Protocol
route	a network route to another machine
host	Any machine on the network
client	The host that wants data or a service
server	The host that provides data or a service

9.1 Network Communication

This section covers network communication from a user point of view. We won't talk much about how to set up hardware or server configurations. This has already been done by your administrator.

9.1.1 Checking Network Status

Sometimes it is unclear if your network is working correctly. There are some easy commands that you can use to check to see if a computer is "alive" on the network. The first and most important command is `ping` which will send a test message to the computer who should respond:

```
[LocalHost]/home/joe:ping groucho
PING groucho (192.168.1.3): 56 data bytes
64 bytes from 192.168.1.3: icmp_seq=0 ttl=255 time=0.7 ms
64 bytes from 192.168.1.3: icmp_seq=1 ttl=255 time=0.3 ms
64 bytes from 192.168.1.3: icmp_seq=2 ttl=255 time=0.3 ms
....
```

This shows that the host **groucho** is alive and well. It provides us with some timing information which indicates the speed of the connection between LocalHost and RemoteHost.

If your DNS (see next section) is not working, you may not be able to ping by name, but perhaps by number. You can also ping by IP number like this:

```
[LocalHost]/home/joe:ping 192.168.1.3
PING 192.168.1.3 (192.168.1.3): 56 data bytes
64 bytes from 192.168.1.3: icmp_seq=0 ttl=255 time=0.7 ms
64 bytes from 192.168.1.3: icmp_seq=1 ttl=255 time=0.3 ms
...etc...
```

If you don't get any response, or do get an error, something is wrong with the network and you will not be able to communicate with that computer.

9.1.2 DNS and Nslookup

The *Domain Name Service* also known as DNS allows you, the users, to translate names like *www.yahoo.com* into a number like 216.32.74.52 which is needed for your computer to communicate over the network. Networks are controlled by gremlins that only use numbers not names. Thus DNS is very important if you are going to use the network at all.

Who provides DNS service? Most of the time your ISP provides this service, especially if you are using a modem, cable modem, DSL, etc. If you are in a large corporate setting, your own company network will provide DNS.

The main tool you need to check that your DNS is working is `nslookup` which can translate names to numbers or vice-versa:

```
[LocalHost]/home/joe:nslookup www.io.com
Server:  flure.pair.com
Address:  209.68.1.159

Name:    www.io.com
Addresses:  199.170.88.21, 199.170.88.41, 199.170.88.39
```

`Nslookup` tells us there are 3 names that go with www.io.com. Large sites like www.io.com and www.yahoo.com often have many numbers because they maintain several servers to handle all the requests to their very busy sites. The server flure.pair.com is our DNS server as seen by our local DNS setup.

DNS can also translate numbers into names:

```
[LocalHost]/home/joe:nslookup 199.170.88.21
Server:  flure.pair.com
Address:  209.68.1.159

Name:    www-02.io.com
Address:  199.170.88.21
```

This was one of the www.io.com sites listed in the first example. If DNS can't find your name, there is little chance you can connect through the internet to it:

```
[LocalHost]/home/joe:nslookup belse.com
Server:  flure.pair.com
Address:  209.68.1.159

*** flure.pair.com can't find belse.com: Non-existent host
```

Sorry, you are SOL. No can do.. (unless you know the number already) Maybe try www.belse.com.

Sometimes you can connect by IP number even when DNS doesn't work. Try to connect to www.io.com with one of the numbers we found. Note that `Linux` has a text browser called `Lynx` that is many-times convenient.

9.1.3 The “R” Commands

The so-called *R-Commands* are `rlogin`, `rcp`, `rsh`, and friends. These are powerful and dangerous commands, and should be used with great care. Many systems exposed to the Internet disable, as they should, these commands because data is completely open to public scrutiny over that network. Now day, some of these commands have secure versions whose data is encrypted, or the commands are replaced by `ssh`, the secure shell.

In order to execute remote commands on a remote machine without passwords, you set up a file called `.rhosts` (in your home dir) that looks like the following:

```
# .rhosts file for RemoteHost
LocalHost joe
```

and on the local machine a complimentary setup allows you to work from there.

```
# .rhosts file for LocalHost
RemoteHost joe
```

Now try to copy a file from here to there:

```
rcp filename RemoteHost:
rcp filename RemoteHost:.            # Same as above.
```

Now copy a file from RemoteHost to LocalHost:

```
rcp RemoteHost:fileA .               # Copy the file to . with same name.
rcp RemoteHost:fileA fileB           # Renames the file
```

Now user `rlogin` to log into RemoteHost:

```
rlogin RemoteHost
Last login: Sat Jan 15 02:53:13 from beppo
You have mail.
[RemoteHost]/home/joe:ls -agl
total 3236
drwxr-xr-x  20 joe users       4096 Jan 14 09:45 .
drwxr-xr-x   5 root     root        4096 Jan 14 08:42 ..
-rw-------   1 joe users       1748 Jan 14 06:59 .ICEauthority
-rw-------   1 joe users        101 Jan 14 08:35 .Xauthority
-rw-r--r--   1 joe users       1506 Jan 14 08:18 .Xdefaults
-rw-------   1 joe users        133 Jan 11 15:56 .bash_history
....etc....
```

9.2 Using X-Windows Over the Network

One great thing about X is that it has networking built right into it. You can use any X application from anywhere on your network. All you need to do is set a few parameters that allow communications.

9.2.1 Connecting X to another client

There are two things you must do to get X to work over a network. First you need to allow other machines to connect to your workstation. One way to do this is to use `xhost RemoteHost`, where RemoteHost is the machine you want to communicate with.

Second, you must telnet (or ssh) to a another workstation, and set the *DISPLAY* environment variable to your workstation like this:

```
[LocalHost]/home/joe:xhost + RemoteHost
[LocalHost]/home/joe:telnet RemoteHost
Red Hat Linux release 6.1
Kernel 2.2.12-20 on an i586
login: joe
Password: ******
Last login: Thu Jan 13 12:13:37 from bistro.rr.com
[RemoteHost]/home/joe:setenv DISPLAY LocalHost:0
[LocalHost]/home/joe:xcalc -title RemoteHost
Ctrl-C
[RemoteHost]/home/joe:exit
[LocalHost]/home/joe:
```

Remember that RemoteHost and LocalHost need to be changed to whatever your workstation is named.

What you should have now is a process that is run by the remote system! It should be clear that this is powerful. In the next chapter we'll talk about other networking features.. If you put a line like the following in your .login or .bashrc, your *DISPLAY* variable will be set automatically for you. Play with this for a second:

```
<tcsh>: setenv DISPLAY ‘who am i | perl -ne ’/\((.+)\)/ ; print $1’“”:0”
or in bash,
<bash>: set DISPLAY= ‘who am i | perl -ne ’/\((.+)\)/ ; print $1’“”:0”
```

9.2.2 Securing Your X Connection

Above we talked about `xhost` and how it allows other machines to connect to your *X-Server* on your workstation. There is a security flaw that allows ALL people from the RemoteHost to connect to your workstation. This "hole" has been used to exploit many a computer.

Fortunately there is a secure alternative to using `xhost` called `xauth`. The way to use `xauth` is to first execute `xauth list` on the RemoteHost, grab the information it produces with your mouse (left click and drag), and then paste it (middle click) on your workstation right after typing `xauth add`.

```
[RemoteHost]/home/joe:xauth list
bistro.marx:0  MIT-MAGIC-COOKIE-1  6c7d8ba30a
bistro/unix:0  MIT-MAGIC-COOKIE-1  7bf452301a230498
localhost:0  MIT-MAGIC-COOKIE-1  a43f0989db
localhost:10  MIT-MAGIC-COOKIE-1  3c08b9241d
```

On your (local) workstation:

```
<bash>: xauth add bistro/unix:0 MIT-MAGIC-COOKIE-1 7bf452301a230498
```

9.3 Network Security for Users

Network security from a user point of view is simple: Don't use easy-to-guess passwords, and don't leave the "R" commands open to attack.

Hard to guess passwords don't resemble any word and should have non alphabetic characters in them. *Hello* is not a good password, but *h3l9Xz* is. Hackers always use `crack` which uses a dictionary to guess passwords, and *Hello* would be cracked in a few seconds.

As far as security is concerned, you need only remember a couple of things.

- Always keep your home directory (˜) protected.
  ```
  <bash>: chmod -R 700 ~/
  ```
- Don't use R-commands over an insecure network, as hackers can see most everything. If you have secure shell `OpenSSH` and secure versions of R-commands, you are better off. Check `http://www.openssh.org` for info on `OpenSSH`.
- Programs like `telnet` show your passwords over the Internet. Use `OpenSSH` from `http://www.openssh.org`
- Never tell anyone your password, and if you do, change it the next day.
- `chmod 600 .rhosts`
- Change your password every few months.
- Never write your password down or identify it as a password.
- Pick an account names and passwords that are difficult to guess. Passwords should have uppercase, lowercase, and numeral characters. Do not use a dictionary word as a password or even part of a password. You can check your password tables agains `crack` at `ftp://ftp.cerias.purdue.edu/pub/tools/unix/pwdutils/crack`.
- Check permissions on files periodically. Most files should be 600 or 700.

Chapter 10

Native Linux Compilers, Software, and Services

Linux includes versions of C++, C, Perl, Python, awk, and many others. It also has text formatting solutions of TEX and LATEX which is what we use to format this document. There are many other freely available additions like Java, Fortran, and Lisp.

Here is what our systems spits out with `man -k compiler`:

```
cccp, cpp (1)        - The GNU C-Compatible Compiler Preprocessor.
f4rpcgen (1)         - an RPC protocol compiler
g++ (1)              - GNU project C++ Compiler
gcc, g++ (1)         - GNU project C and C++ Compiler (egcs - 1.1.2)
gcc, g++ (1)         - GNU project C and C++ Compiler (egcs-1.1.2)
tic (1m)             - the terminfo entry-description compiler
zic (8)              - time zone compiler
B (3)                - The Perl Compiler
B::Bytecode (3)      - Perl compiler's bytecode backend
B::C (3)             - Perl compiler's C backend
B::CC (3)            - Perl compiler's optimized C translation backend
B::Deparse (3)       - Perl compiler backend to produce perl code
O (3)                - Generic interface to Perl Compiler backends
```

Here is what our systems gives for `man -k language`:

```
Tcl (n)              - Summary of Tcl language syntax.
bc (1)               - An arbitrary precision calculator language
gawk (1)             - pattern scanning and processing language
perl (1)             - Practical Extraction and Report Language
perlxs (1)           - XS language reference manual
python (1)           - an interpreted, object-oriented programming language
```

Consult the following websites for current software information:

`http://www.freshmeat.net` for Linux software.
`http://www.sourceforge.net` for OpenSource software.
`http://www.newsforge.com` Open Source News.
`http://www.seul.org` Linux software for education and science.
`http://www.slashdot.org` Don't forget Slashdot.

Bibliography

[1] Welsh, Dalheimer, and Kaufman, *Running Linux,* 3rd *Edition. O'Reily & Associates*, 1999.

[2] Ellen Siever, *Linux in a Nutshell,* 3rd *Edition. O'Reily & Associates*, 2000.

[3] M. Sobell, *A Practical Guild to Linux. Addison-Wesley*, 1997.

[4] `http://www.linuxdot.org/nlm/` *Linux Newbie Guide.*

[5] `http://sunsite.auc.dk/linux-newbie/` *Linux Newbie Administrator Guide.*

[6] The AfterStep Homepage at `http://www.afterstep.org`

Appendix A

Glossary

alias	A shortcut name for another command
ASCII	American Standard Code for Information Interchange = text
background	Processes work silently here and dont bother you
bash	The GNU Bourne-Again SHell.
bin	Refers to an executable "binary", usually a directory
cwd	Your Current Working Directory
cursor	The active point for editing, usually a highlighted rectangle
daemon	A program in the background and provides a service, like HTTPD
device	A physical or logical component of a computer
directory	A folder in the computer
DNS	Domain Name Service: domain names for IP numbers & vice-versa
environment	A set of parameters your shell works in
execute	To enter a command
ext2	The native Linux filesystem
file permissions	They control who reads, writes, and executes files
foreground	Where processes with your console
FTP	File Transfer Protocol, used to move files over networks
GID	Group ID of a file or process
GUI	Graphical User Interface. Think Windows, but better!
$home	Your home directory
host	A computer on a network
job	A process running in your shell
kernel	The core software that runs Linux
man	The MANual pages
mount	Attach a disk partition to Linux
mount point	An empty directory you attach (mount) a partition to
network	A series of communication channels for computers
partition	A subdivision of a hard disk. A slice of pie.
path	/This/is/where/you/are/ in the filesystem (think folder)
PID	Process ID
priority	Number indicates how important a process is to the kernel
process id	Every running program has such a number
prompt	That piece of text just before you type commands. Customizable.
regular expression	A pattern used for searching text.
route	The path data takes through a nework.

script	**An shell program much like a ”batch file” in MS-DOS.**
shell	**A program that reads/executes command from users.**
server	**A computer that provides data and/or services**
stderr	**Standard Error**
stdin	**Standard Input**
stdout	**Standard Output**
string	**Just a text word**
swap	**Memory on a disk**
tcsh	**The C-Shell, very friendly**
UID	**User ID. Each user has one and files get tagged with it.**
username	**Your login name**
user	**A computer user**
xterm	**A console program for X-Windows**