Kernel vs. Operating System and Tools

Kernel

- The word **Linux** is often sloppily applied to the entire operating system and environment on computers which are equipped with a complete Linux distribution; but in fact, there are quite a few components which are necessary in order to have a fully functional platform
- Narrowly defined, Linux is only the **kernel** of the operating system (OS)
  - The kernel is the central component that connects the hardware to the software, and manages the system's resources, such as memory, CPU time sharing among competing applications and services
  - It handles all the devices that are connected to the computer by including so-called device drivers, and makes them available for the operating system to use
- A system running only a kernel has limited functionality, and the only place you will see that is in a dedicated device (often termed an **embedded device**) such as inside an **appliance**

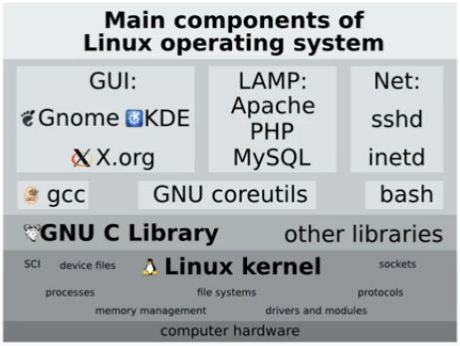
Operating System

In order to do something useful and to be able to do a variety of things as needs arise, you need some other components, which in and of themselves are not strictly part of Linux:

- Important **system libraries**
  - Usually these are **shared libraries** or **dynamic linked libraries**
  - They can be used simultaneously by more than one program
  - The most important one is **libc**, which is used by virtually every application (among other things it handles the communication between the applications and the kernel)
- Important **system services** (sometimes called **daemons**)
  - Started when the system runs to control and monitor activities on the system, e.g. networking, printing, disk maintenance, noticing when new equipment is plugged in, monitoring system load and performance, etc.
- Basic **system utilities**
  - Those which handle listing files, viewing them, renaming them, removing them, etc.; bringing network connections up and down; compressing and decompressing files, etc.
  - Many of these utilities (which are often simple ones) are needed by the services already mentioned
  - A particularly important program is the command **shell** program, which is what users interact with when they work at a command line, but which is also used by non-interactive scripts
  - The default shell for Linux is usually **bash**, which stands for **B**ourne **A**gain **SH**ell, since it is an extension of the older **sh**, or Bourne Shell program

Main Components of Linux Operating System

Strictly speaking, the ingredients we discussed (kernel, including device drivers, services, and utilities) are enough to constitute a complete operating system, but other ingredients will be found on general service computers, as what we presented will only give you a **command-line based terminal**.



Graphical User Interface (GUI)

- Normal users will almost always be running a **Graphical User Interface** (GUI)
- Almost all desktop Linux systems will be built using the **X Window System** (or **X**) as the base of this interface; it has been around since at least 1984
- Besides **X**, there will be a so-called **window manager** which controls the appearances and behaviour of windows
- And a **desktop manager** which controls the entire desktop; the two most common choices in Linux are **GNOME** and **KDE**

Add-on Applications

- Many applications are part of the standard installation, but they are not part of the operating system - they are added on
  - The early versions of the Windows platform didn't contain many basic utility programs, and these had to be supplied by third-party providers until built-in implementations were deployed
  - The same is true of Internet browsers
- While many operating systems have now gotten so dependent on some of these programs being around that it is hard or impossible to operate without them, strictly speaking they are not absolutely required
- One advantage of Linux is that since there is so much choice for these components, they are used in a **modular way**, and the system can run with different or even no choices for them, unlike strictly controlled commercial platforms which either do not give such freedoms, or work hard to obscure and make it difficult to exercise options

Developmental Environment Tools

- Another important component is the **developmental environment tools**, such as compilers, debuggers, etc.
- While not all users need these, they are always available in Linux distributions

Linux Distribution

- It is the role of the **Linux distribution** to bring all these ingredients together in a coherent way
- Since they all do this somewhat differently, the look and feel can be quite dissimilar even when everything is built on top of the same Linux operating system kernel

UNIX and Linux

The Differences

- Linux is only the kernel (everything else that makes up the full operating system is drawn from a number of sources); and it is *not* UNIX, although it's clearly UNIX-like
- UNIX and Linux have had very different evolutions:
  - UNIX was developed about 1969 by Thompson, Canaday, and Ritchie, and from the very outset it was designed to be a serious enterprise operating system
  - It grew up largely outside of the Intel family of CPUs, although it was later ported to it
  - By the time Linux first appeared in 1991, UNIX had already become quite fractured: there were many varieties, grouped in two major families; **System V** arising from the original code at Bell Labs, and **BSD**, arising from the University of California at Berkeley
  - Linux, on the other hand, began as a toy operating system only on the x86 architecture; it is doubtful anyone had any idea of how robust it would become or how many architectures it would wind up supporting

UNIX Variants

- Sorting out the fractious history and differences among the different UNIXs would be a lengthy task, but by 1991 there were many variants, often tied to a specific hardware platform and vendor:
  - SGI had IRIX
  - Sun had SunOS and Solaris
  - IBM had AIX
  - Hewlett Packard had HPUX
  - Cray had UNICOS
  - DEC had Ultrix
- Each one of these manufactures often had several varieties running even on its own universe of hardware
- **SCO** was one of the only variants not arising from a hardware company

UNIX Variants (Cont.)

- While there were various efforts to achieve some standardization, most vendors had strong self-interest in keep in things proprietary
- As a minimum there were always two major flavors to be considered, **System V** and **BSD**, and their behavior could be quite different even on quite basic matters such as signal handling
- Application developers interested in portability had to resort to the use of ugly **`#ifdef`** statements
- Even where the APIs were not that different, the actual implementation could be radically different from platform to platform
- Basic kernel architectures also differed
- Each platform had its own set of basic file utilities, shells, etc.

GNU

- The **FSF**'s (Free Software Foundation) **GNU** (GNU's not UNIX) project developed freely-distributable versions of many basic utilities, such as **tar**, **ls**, **grep**, etc., and even more important, the **gcc** compiler and the basic C-library, **libc**
- Linux could not have been born or grown without the availability of tools from the GNU project
- We will stay out of the arguments about whether Linux should properly be loosely called GNU/Linux or something similar, but just note that what is often sloppily called Linux in fact contains many GNU components; properly speaking, Linux is only the kernel

Correlations

- While UNIX and Linux are not the same thing, Linux has always borrowed heavily from UNIX
- Most basic components of Linux, such as an **inode**-based filesystem, accessing hardware through device nodes, multi-process scheduling, process creation and destruction, are completely rooted in UNIX
- This is because the developers of Linux have always had a good footing in the UNIX world, and because of the availability of the UNIX tools from GNU and other non-GNU open-source projects
- Whenever possible, Linux has tried to accommodate both major variants of UNIX as far as the API is concerned; striving for POSIX type behavior is above that level

Correlations (Cont.)

- As such it has never been very difficult to port UNIX applications to Linux, unless the application has relied very heavily on certain idiosyncrasies of a particular UNIX implementation
- The open nature of the Linux development model has thus far avoided the serious fracturing that took place in UNIX
- It is perhaps ironic that the easily legal possibility of having Linux fork into competing versions at any time is perhaps what has prevented it
- Many hardware vendors now seriously support Linux on their hardware and the long range future of their own versions of UNIX is often in doubt
- It could well be that the Linux plan for world domination is inevitable; at least as far as killing off other UNIX-like operating systems

Linux Standard Base

There are a variety of different standards (specifications) that are relevant to working on UNIX-like operating systems such as Linux. For example, a typical **man** page (we will discuss **man** pages and other help and documentation utilities later), such as the one for **open()** in the following example, might have a statement in its **CONFORMING TO** section, such as:

```
1  CONFORMING TO
2         SVr4,  4.3BSD, POSIX.1-2001.  The O_DIRECTORY, O_NOATIME, O_NOFOLLOW, and
            O_PATH flags are
3         Linux-specific, and one may need to define _GNU_SOURCE (before including
            any header files)
4         to obtain their definitions.
5
6         The O_CLOEXEC flag is not specified in POSIX.1-2001, but is specified in
            POSIX.1-2008.
7
8          O_DIRECT  is  not  specified in POSIX; one has to define _GNU_SOURCE
             (before including any
9          header files) to get its definition.
```

The history of the different standards and specifications is complicated and is much wrapped up in the convoluted UNIX family tree with its two main branches, BSD and System V. For the most part, Linux strives to be POSIX-compliant, a later standard that incorporates earlier ones and stands for Portable Operating System for UNIX. Note, however, Linux distributions are not generally certified as UNIX-compliant, as the rapid development pace is not amenable, the cost is high, and the derived benefits would generally be considered marginal.

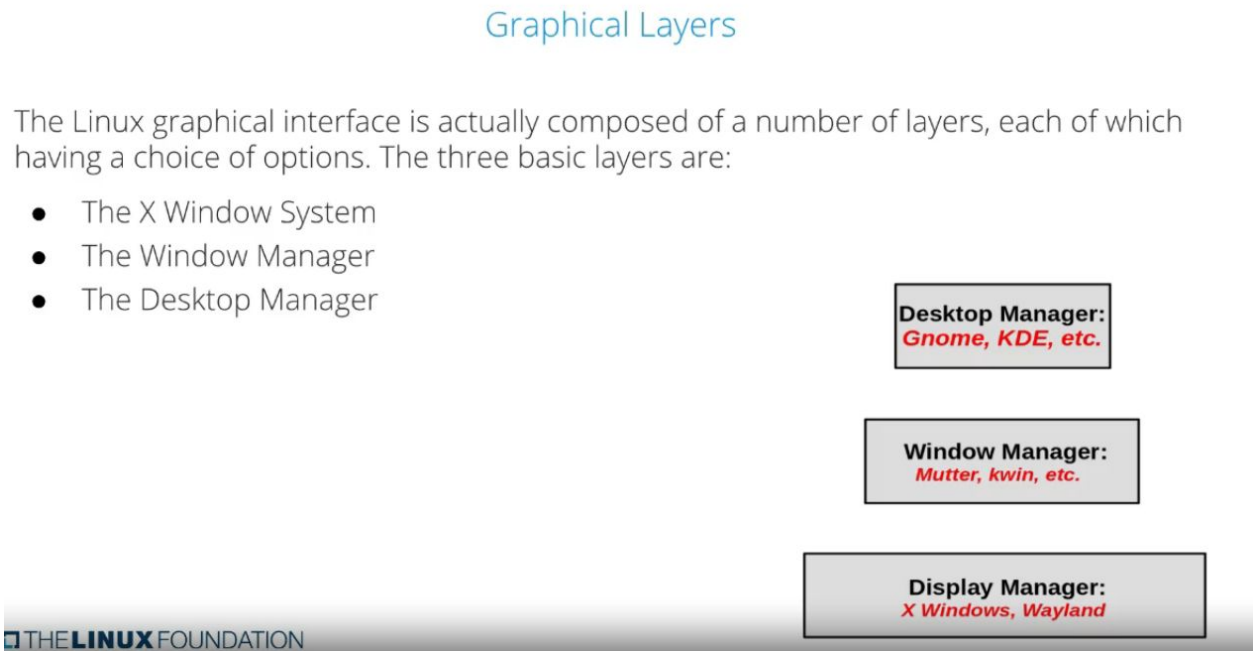
It is important, however, that software developed for Linux be portable across different distributions without much pain, and the LSB specification (Linux Standard Base) has stepped into that role. It is administered by The Linux Foundation.

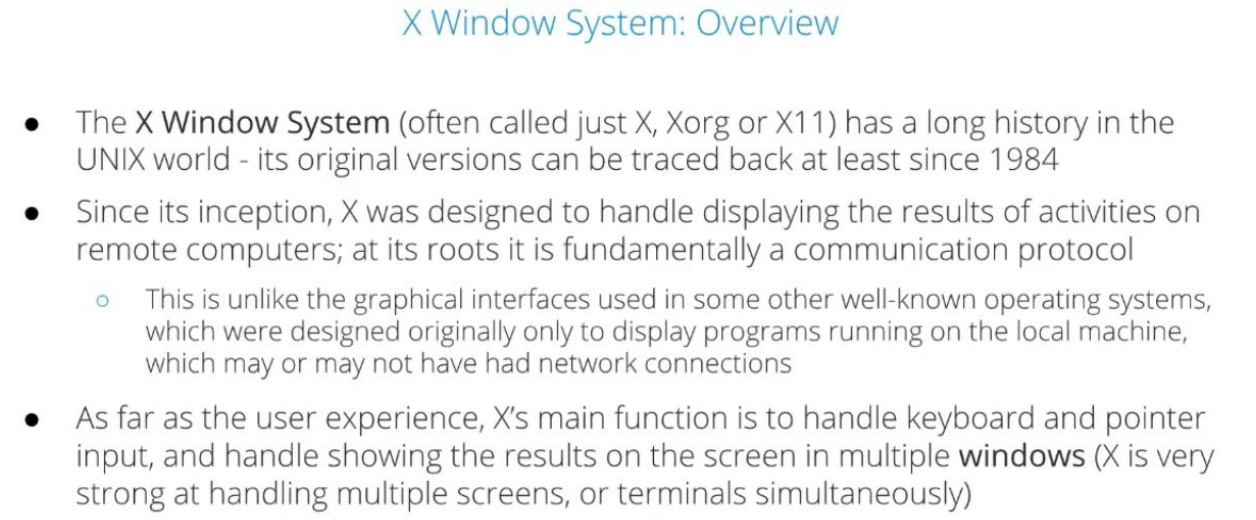
In addition to coding considerations, the LSB also considers matters such as standard utilities and libraries required, where various things are put on the filesystem, etc. If one follows the specification, one should be able to develop on any distribution and use on any other if both are LSB-compliant.

Note - Most of the material is written with the three main Linux distribution families in mind:

- Red Hat/Fedora
- openSUSE/SUSE
- Debian.

Graphical Layers and Interfaces





Nomenclature

- In X nomenclature,
  - The **server** is what runs on your local machine which handles the input and display
  - The **client** is the application being displayed and is as likely to be anywhere on the network and worldwide Internet as it is on the local machine
- This differs from the usual server/client distinction you may be used to, where the server is a remote machine, and the client the local machine

Criticism

- A criticism of X sometimes heard is that it has a high inherent overhead because it works on a network paradigm, which also leads to a complicated design
- While this might have been true with early versions, modern implementations make very efficient use of **unix domain sockets**, **shared memory** and other optimizations, and the criticism does not hold weight anymore

Configuration

- There are many standard programs that ship with X, but the normal user will probably never have to interact with them directly, but will adjust properties and preferences through the graphical interface
- Almost all Linux distributions use the version of X that is supplied by X.org
  - The basic configuration file can be found at **`/etc/X11/xorg.conf`** and controls things like color-depth, display resolution, scanning rates, what pointers are available, which video card to use, etc.
  - If you are lucky, you will never have to touch **`xorg.conf`** directly; it is best to let it be generated by the installation program and then subsequently tweaked through graphical interfaces
  - In most modern Linux distributions this file has actually disappeared, and X is auto-configured on system start, although one can always substitute a custom file; sometimes, manual intervention is required (e.g. if you have recent or unusual hardware)

Wayland

- X is a rather old system; it dates back to the mid 1980s and as such has certain deficiencies on modern systems (with security for example) as it has been stretched rather far from its original purposes
- A newer system known as **Wayland** is expected to gradually supersede it and was adopted as the default display system in Fedora 25

Window Managers: Overview

- By itself, X has only limited functions; it does not control the exact placement and appearance of windows in the graphical interface
- This is the job of the **window manager**; some of its functions include:
  - Controlling the appearance of a window
  - Controlling pointer focus properties
  - Handling multiple desktops
  - Providing tabbed windows
  - Controlling visual effects
- This is not a complete list, and the line between window manager and desktop manager is not always well-defined
- Also, the window manager itself gives you the capability of altering many of these properties and different window managers can be tweaked to look very much alike

Window Managers for Linux

- There are a number of window managers available for Linux, and desktop managers have default choices:
  - For GNOME 3, the default is **mutter**
  - For KDE it is **kwin**
  - Other ones in use include **metacity**, **fvwm**, **fluxbox**, **enlightenment**, **sawfish**, and **xfwm**
- Some of the alternatives are very flashy, while some (such as **fvwm** and **fluxbox**) are very minimal, very fast, and work beautifully on limited hardware

Desktop Managers: Overview

- The **desktop manager** sits above X and the window manager; it is what the user is most likely to directly interact with
- The tasks of the desktop manager include:
  - Providing taskbars, menu bars, drop-down menus, and methods of configuring them
  - Offering applications (e.g. clocks, performance monitors, volume controls, etc.)
  - Enabling placing icons and program launchers on the desktop and/or the taskbar
  - Giving choices for themes, desktop backgrounds and wallpaper, screensavers, etc.
  - Providing methods for drag and drop functionality, etc.
  - Making it possible to save the desktop state from one session to another subsequent login

Desktop Managers for Linux

- The most common desktop managers in use for Linux are:
  - **GNOME** (heavily dependent on the **gtk** set of graphical libraries)
  - **KDE** (built upon the **QT** libraries)
  - Others exist including **XFCE**
- Furthermore, some of the lightweight window managers, such as **fluxbox** and **fvwm** do not even require a desktop manager, as they have already have enough functionality to survive on their own
- Many distributions give you a choice of selecting one or the other desktop during installation; some let you choose both at installation or let you install the other at a later point
- Then you can choose which desktop manager you would like when you login

GNOME and KDE

- It is entirely possible to run KDE programs when running the GNOME desktop and vice versa, as long as the underlying libraries are installed; you can even drag and drop between the two, as the developers have tried hard to maintain interoperability
- If your distribution has done a sensible job of controlling dependencies on installing programs and their requisites, this should all be transparent to the user
- Because each window manager is so flexible, they can be made to look very much the same; for example, on RHEL systems, the default themes for GNOME and KDE look so much alike it can be hard to tell which one you are actually using

Terminal Window Options

- Linux users often take advantage of the command line to accomplish tasks
- While one can always bring up a one shot launch box (usually by hitting **`Alt-F2`** in most desktop managers), one needs a **terminal window** program to open up a window that can be used repeatedly
- The GNOME desktop contains the **gnome-terminal** program, which is very full-featured; it can be always be invoked from menus, which vary slightly between distributions, and also added to the *Favorites* panel
- If the **nautilus-open-terminal** package is installed on any but some of the most recent GNOME-based distributions, you can always open a terminal by right clicking anywhere on the desktop background and selecting *Open in Terminal* (this may work even if the package is not installed)

Terminal Window Options (Cont.)

- Menu items on the top bar make the presentation extremely flexible; right clicking anywhere in the terminal also provides configuration controls
- Extremely useful is the opportunity to use multiple tabs as in browsers, within the same terminal
- The KDE desktop contains the **konsole** program, which is similarly flexible; opening new tabs is termed as opening shells
- Desktop managers generally have a preferred applications menu choice in which you can pick the default terminal application
- The full-featured terminals, such as **gnome-terminal** and **konsole**, can be slower to display many lines of text, as compared to more ancient programs such as **xterm** or **rxvt**; however, in most modern systems this will not be noticeable, and the enhancements are well worth it

man Pages

man is the workhorse of Linux documentation, as it has been on all UNIX-like operating systems since their inception. Its name is short for manual. In fact, the first edition of the UNIX Programmer's Manual was released in 1971 and was probably the first case of online documentation. It is most often invoked from the command line in a terminal window.

When you invoke man at a terminal window, it will automatically pipe its output into your pager, which on most Linux systems is **less**; on older systems, it may be **more**. You can change this by altering the value of the **PAGER** environment variable. You will notice that man pages are referenced by chapter number, e.g. **gzip(1)** is in the first chapter. What chapter a given man page belongs in depends on its subject:

Chapter	Description
1	User commands (standard commands)
2	System calls
3	Subroutines (library functions)
4	Devices
5	File formats, and files used by a program
6	Games
7	Miscellaneous
8	System administration
9	Kernel documentation
n	New, mainly used by Tcl/Tk

In addition, there are sometimes chapters with a **p** or **x** suffix, such as **1p**, or **3x**, where the **p** stands for the **POSIX** standard specifications, and **x** stands for X Window System documentation. Other subsections may be present as well on your system.

One complication is that many keywords can have more than one man page. For instance, **socket** has at least two different man pages, in chapters 2 and 7. You can look at any one of them by specifying the particular chapter, as in:

```
$ man 7 socket
```

or you can see all of them in sequence by doing:

```
$ man -a socket
```

man has a lot of options (do **man man**), some of which have utility short hand forms. For example, by doing either of these commands:

```
$ whatis socket
$ man -f socket
```

you will get a list of all man pages that have socket in their name. Likewise, if you do either of the commands:

```
$ apropos socket
$ man -k socket
```

you will get a list of all man pages that discuss sockets, whether or not it is in their name.

Reading info

info is a simple-to-use documentation system, hypertextual in nature, although it does not require a graphical browser. The documentation is built using the Texinfo system, which is a reader you need know nothing about.You may have noticed that many man pages say you can get more detailed information by running info on the command, such as in:

E.g - $ info ls

This pops you into a documentation page, which you can page through by hitting the space bar. Other key bindings are:

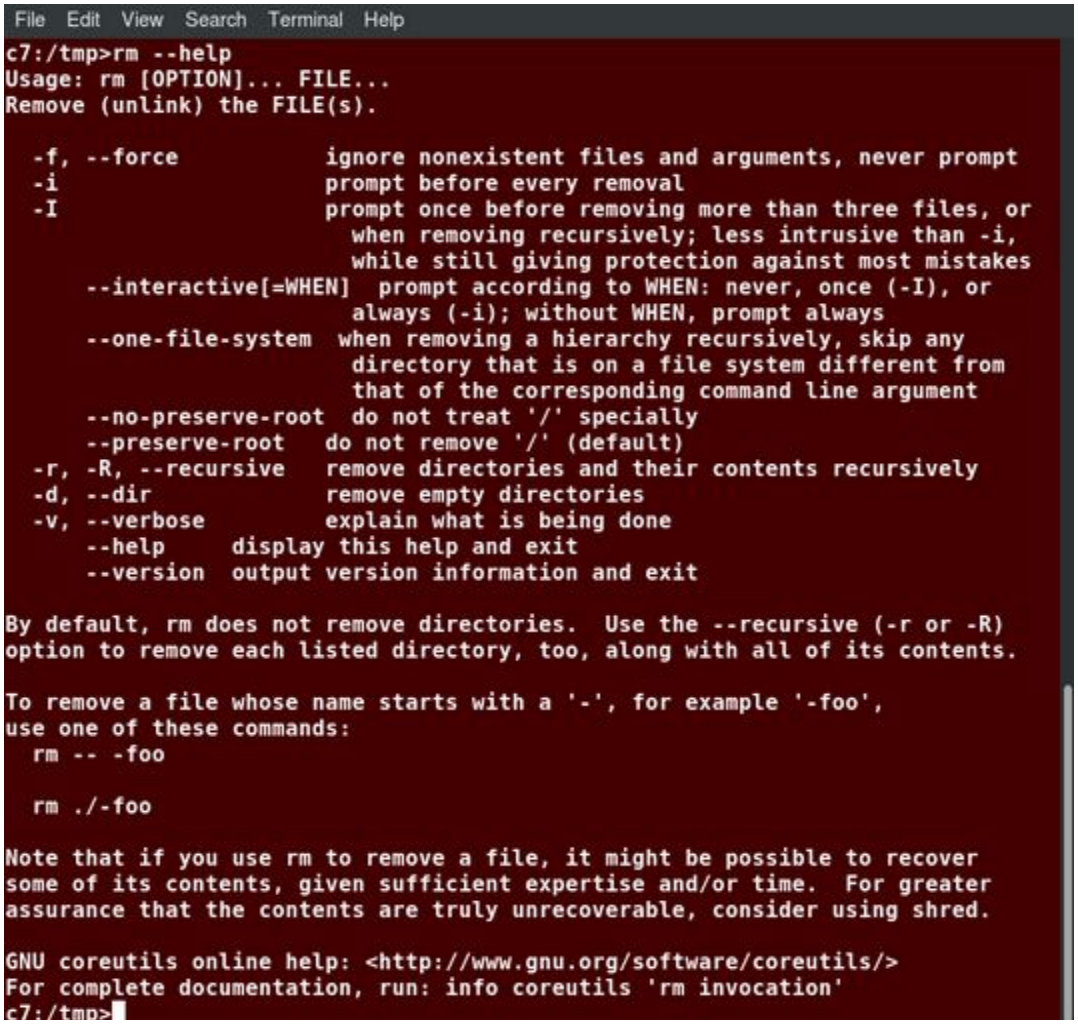
Key	Function
n	Go to next node
p	Go to previous node
u	Go to upper node
l	Go to last visited node
space or PageDown	Go to next page
delete\verb or backspace or PageUp	Go to previous page
/ or CTL-s	Search for the string prompted for
i	Search for a node containing the string prompted for

You will notice we distinguish between nodes and pages. The **info** page is a hierarchical tree of nodes, each of which may have multiple pages, or screenfuls.

Lines which begin with an asterisk (*) are nodes which you can jump to by moving to that line (with the arrow keys for example) and then hitting return. **info** is easier to use than it is to describe. It often contains more exhaustive information than the man page, but sometimes it just regurgitates it.

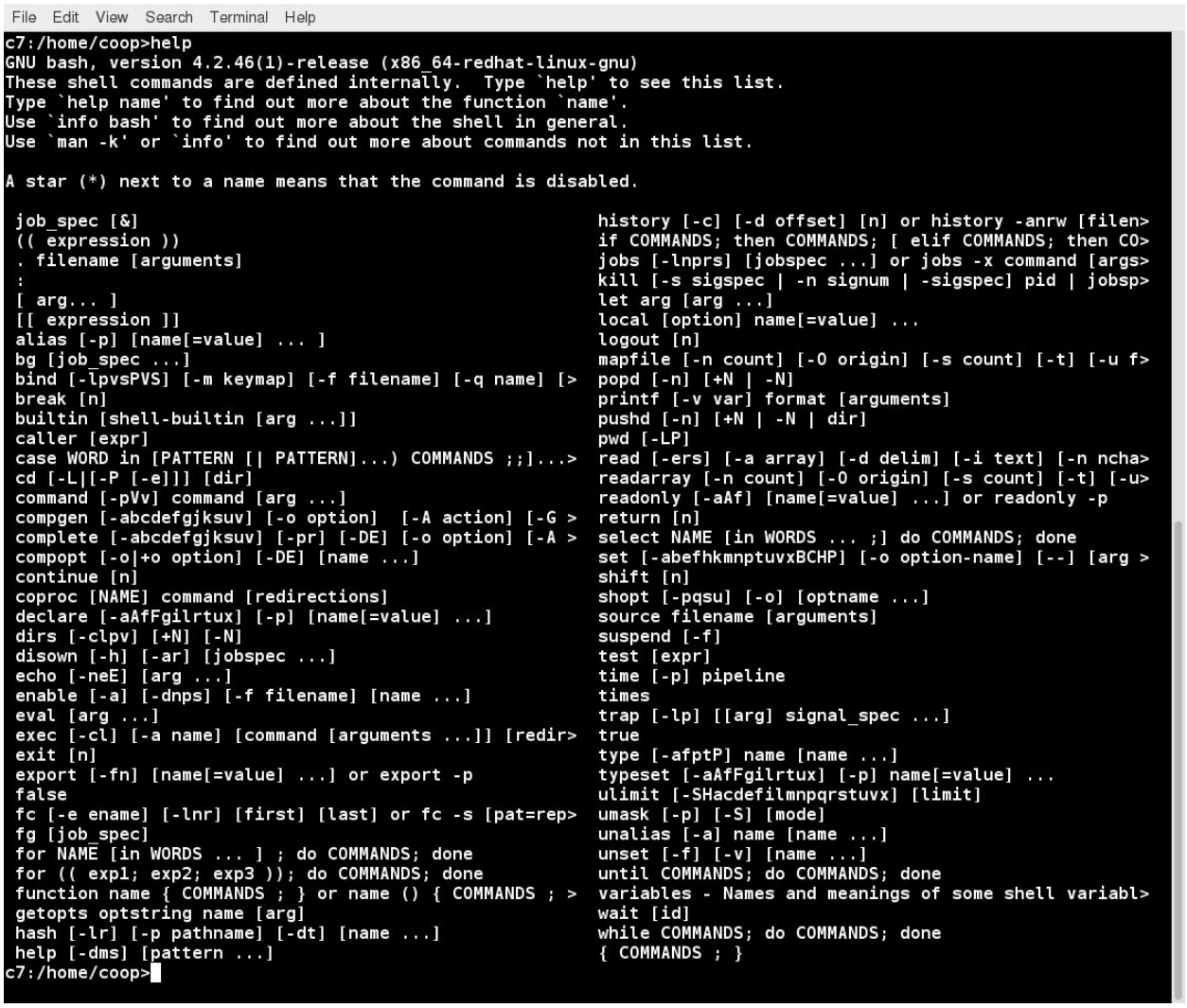
Reading --help and help

Many of the commands on your system will generate a brief discussion of usage and options if you run them with the **--help** option. For example, trying this with **rm** by doing **rm --help** gives the output seen in the screenshot below.



This is often all you need and can be consumed much quicker than running man or info. There is also a **help** command, which is actually part of the bash shell, and only gives information

about commands which are actually part of the shell itself. Typing **help** by itself generates the screenshot shown below -



and information on a particular command can be done as in:

```
$ help pwd

pwd: pwd [-LP]
    Print the current working directory.  With the -P option, pwd prints
    the physical directory, without any symbolic links; the -L option
    makes pwd follow symbolic links.
```

It is important to note that there are programs which have two incarnations, one in the bash shell and one as a standalone program. For example, these two commands are similar but not identical:

```
$ echo hello
$ /bin/echo hello
```

By default, the command built into the shell is invoked, rather than the one in the path. Likewise, the results of **man echo** and **help echo** are not the same. This can be confusing.

Multiple man Pages

Often, there are multiple man pages (in different chapters of the manual) for a given topic. As an example, we will give **stat**.

If you just do:

```
$ man stat
STAT(1)                         User Commands                         STAT(1)

NAME
       stat - display file or file system status

SYNOPSIS
       stat [OPTION] FILE...

DESCRIPTION
          Display file or file system status.

.....
```

you will get a description of the command line utility.

Try doing:

```
$ man -k stat
```

which is equivalent to doing **apropos stat**. You will get a long list of man pages that reference the string **stat**.

To narrow the list down to pages that are an exact match, do:

```
$ man -f stat
```

and then you can get the page from chapter two by doing:

```
$ man 2 stat
```

If you want to see all the pages, you can do:

```
$ man -a stat
```

and then you can see each of the relevant chapters in turn, typing **q** after each one is displayed to proceed the next one.

Introduction to Text Editors

Text Editors

- At some point you will have to edit text files, and while graphical system administration applications can help you avoid much of this, often it is far more laborious this way than it is to directly work on relevant files with a **text editor**
- By now you have realized Linux is full of choices, and when it comes to text editors, the inventory of alternatives is enormous; the editors vary from simple to very complex
- Many of these editors are already familiar to developers used to working on UNIX-like systems
- Venerable stalwarts such as **vi** and **emacs** are completely compatible with the versions used on other operating systems

Available Editors

Here is a partial list of widely available editors on any Linux distribution:

- **vi** (standard editor available on all UNIX-like systems
- **vim-enhanced** (vi with many enhancements)
- **vim-X11** (vi with full graphics support)
- **emacs** (standard editor available on all UNIX-like systems)
- **xemacs** (variant of emacs preferred by some)
- **nano** (small and easy to use)
- **gedit** (graphical editor part of the GNOME desktop)
- **KWrite** (graphical editor part of the KDE desktop)
- **kate** (graphical editor part of the KDE desktop)
- **nedit** (simple graphical editor)

Using echo and cat

If you just want to create a file, without even doing an editor, there are two standard ways to just create one from the command line and fill it with content.

The first is to just use **echo** repeatedly:

```
$ echo line one > myfile
$ echo line two >> myfile
$ echo line three >> myfile
```

This sort of game is often played from scripts.

A second way is to use **cat** combined with redirection:

```
$ cat  << EOF > myfile
> line one
> line two
> line three
> EOF $
```

either of which produces a file which has in it:

```
line one
line two
line three
```

About vi Editor

vi

- **vi** (pronounced as "vee-eye") can be found on any Linux system; usually the actual installed program is **vim** (**vi Improved**) which is aliased to the name vi
- Even if you do not want to use vi, it is good to gain some familiarity with it as it is such a standard tool, and there may be times when you have no other choice
- GNOME offers a very graphical interface known as **gvim** and KDE offers **kvim**; both may be found to be easier to use at first
- Using the basic vi (or vim) program, all commands are entered through the keyboard; you do not have to deal with the mouse, unless you are using a graphical version
- The most confusing part about vi is that it has two fundamental modes: **command** and **insert**, and you always have to remember which mode you are in; you switch from one to the other by hitting the escape key, and keyboard functions in each mode can be quite different

Starting, Exiting, Reading and Writing Files in vi

Command	Description
vi myfile	Start **vi** and edit **myfile**
vi -r myfile	Start **vi** and edit **myfile** in recovery mode from a system crash
:r file2<RET>	Read in **file2** and insert at current position
:w<RET>	Write out the file
:w myfile<RET>	Write out the file to **myfile**
:w! file2<RET>	Overwrite **file2**
:x<RET> or :wq<RET>	Exit **vi** and write out modified file
:q<RET>	Quit **vi**
:q!<RET>	Quit **vi** even though modifications have not been saved

Searching for Text in vi

Command	Description
/pattern<RET>	Search forward for pattern
n	Move to next occurrence of search pattern
string<RET>	Search backward for pattern
N	Move to previous occurrence of search pattern

Changing Position in vi

Command	Description
arrow keys	Use the arrow keys for up, down, left and right; or:
j or **<RET>**	One line down
k	One line up
h or Backspace	One character left
l or Space	One character right
0	Move to beginning of line
$	Move to end of line
w	Move to beginning of next word
b	Move back to beginning of preceding word
:0 <RET> or **1G**	Move to beginning of file
:n <RET> or **nG**	Move to line n
:$ <RET> or **G**	Move to last line in file
^f or **PageDown**	Move forward one page
^b or **PageUp**	Move backward one page
^l	Refresh and center screen

Changing, Adding and Deleting Text in vi

Command	Description
a	Append text after cursor; stop upon **Escape** key
A	Append text at end of current line; stop upon **Escape** key
i	Insert text before cursor; stop upon **Escape** key
I	Insert text at beginning of current line; stop upon **Escape** key
o	Start a new line below current line, insert text there; stop upon **Escape** key
O	Start a new line above current line, insert text there; stop upon **Escape** key
r	Replace character at current position
R	Replace text starting with current position; stop upon **Escape** key
x	Delete character at current position
Nx	Delete **N** characters, starting at current position
dw	Delete the word at the current position
D	Delete the rest of the current line
dd	Delete the current line
Ndd or **dNd**	Delete N lines
u	Undo the previous operation
yy	Yank (cut) the current line and put it in buffer
Nyy or **yNy**	Yank (cut) N lines and put it in buffer
p	Paste at the current position the yanked line or lines from the buffer

About emacs

emacs

- **emacs** is available on all Linux systems; a less common variant, **xemacs**, is also generally available
- emacs has many other capabilities other than text editing: it can be used for email, debugging, etc.
- Rather than having different modes for command and insert, like vi, emacs uses the **Control** and **META** key for special commands
  - You can use `Alt` for the META key or you can use `Escape`; if you use `Escape`, you must release the key before you type the rest of the key combination or commands listed further in this lesson
  - It is purely a question of taste as virtually all keyboards today have both an `Alt` and `Escape` keys

Starting, Exiting, Reading and Writing Files in emacs

Command	Description
emacs myfile	Start **emacs** and edit **myfile**
Ctl-x i	Insert prompted for file at current position
Ctl-x s	Write out the file keeping current name
Ctl-x Ctl-w	Write out the file giving a new name when prompted
Ctl-x Ctl-s	Write out all files currently being worked on and exit
Ctl-x Ctl-c	Exit after being prompted if there any unwritten modified files

Searching for Text in emacs

Command	Description
Ctl-s	Search forward for prompted for pattern, or for next pattern
Ctl-r	Search backwards for prompted for pattern, or for next pattern

Changing, Adding and Deleting Text in emacs

Command	Description
Ctl-o	Insert a blank line
Ctl-d	Delete character at current position
Ctl-k	Delete the rest of the current line
Ctl-_ or **Ctl-x u**	Undo the previous operation
Ctl-space	Mark the beginning of the selected region; the end will be at the cursor position
Ctl-w	Yank (cut) the current marked region and put it in buffer
Ctl-y	Paste at the current position the yanked line or lines from the buffer

Changing Position in emacs

Command	Description
arrow keys	Use the arrow keys for up, down, left and right; or:
Ctl-n	One line down
Ctl-p	One line up
Ctl-f	One character left
Ctl-b	One character right
Ctl-a	Move to beginning of line
Ctl-e	Move to end of line
M-f	Move to beginning of next word
M-b	Move back to beginning of preceding word
M-<	Move to beginning of file
M-x goto-line n	Move to line n
M->	Move to end of file
Ctl-v or **PageDown**	Move forward one page
M-v or **PageUp**	Move backward one page
Ctl-l	Refresh and center screen

Shells

A shell is a command line interpreter which can constitute the user interface for terminal windows. It can also be used as a mechanism to run scripts, even in non-interactive sessions without a terminal window, as if the commands were being typed in.

For example, typing:

```
1  $ find . -name "*.c" -ls
```

at the command line accomplishes the same thing as running the following script:

```
1  #!/bin/bash
2  find . -name "*.c" -ls
```

The **#!/bin/bash** at the beginning of the script should be familiar to anyone who has developed any kind of script in UNIX environments. Following the magic **#!** characters goes the name of whatever scripting language interpreter is tasked with executing the following lines. Choices include **/usr/bin/perl**, **/bin/bash**, **/bin/csh**, **/usr/bin/python** and **/bin/sh**.

Linux provides a wide choice of shells; exactly what is available to use is listed in **/etc/shells**; e.g. on one system we get:

```
1  $ cat /etc/shells
2  /bin/sh
3  /bin/bash
4  /sbin/nologin
5  /bin/tcsh
6  /bin/csh
7  /bin/ksh
8  /bin/zsh
```

Most Linux users use the default bash shell, but those with long UNIX backgrounds with other shells may want to override the default. It is worth reviewing the main choices in the historical order of introduction.

More on Shells

- On most Linux systems, sh is just a link to bash, scripts which are invoked as sh will only work without the bash extensions; a similar relationship exists between csh and tcsh
- Maximum portability is obtained by writing scripts that use only the older features, so you will see many scripts using sh in place of bash; but these days bash is certainly present on all Linux systems, and indeed on almost all UNIX system
- Porting of scripts between sh, ksh and bash is relatively easy and straightforward
- Porting from csh variants is more complicated because of the quite different syntax
- Linux systems make all of these shells available so porting is generally optional
- bash and ksh offer enhanced readability of scripts as compared to sh due to extensions; they also tend to run somewhat faster because the original philosophy of sh was to be minimal, have few internal commands, and rely on external commands for simple operations such as **`echo`**
- The newer shells replace some of these external commands with built-in versions
- While this is more efficient, sometimes subtle differences can wreck scripts
- Some of these snags can be avoided by using the full path in the scripts to the external command, such as **`/bin/echo`** instead of just **`echo`**, which would not search the path

Any command shell can be invoked merely by typing its name at the command line. A user's default shell can be changed with the **chsh** utility.

We will concentrate on bash, which is generally the default shell under Linux.

Kinds of shells

- A login shell is one requiring a password (logging in)
- An interactive shell is one in which the standard input/output streams are connected to terminals
- A non-interactive shell is one in which the standard input/output streams may be connected to a process, etc.

Initialization

Interactive shells

Login shells:

- if **/etc/profile** exists, source it
- if **~/.bash_profile** exists, source it
- else if **~/.bash_login** exists, source it
- else **if ~/.profile** exists, source it
- on exit, if **~/.bash_logout** exists, source it

Non-login shells:

- if **~/.bashrc** exists, source it

Non-interactive shells

Despite what the man page says, it seems to be the same as interactive shells.

Note that by default, most distributions include a system-wide file (usually **/etc/bashrc**) from the user's **~/.bashrc**.

Aliases

Aliases permits custom definitions. Typing **alias** with no arguments gives the list of defined aliases. **unalias** gets rid of an alias.

Some alias examples are shown below:

```
alias l='ls -laF'
alias dir='ls -latF'
alias rm='rm -i'
alias mv='mv -i'
alias cp='cp -ipdv'
alias df='df -T'
alias myyahoo='firefox https://my.yahoo.com'
alias diffside='diff --side-by-side --ignore-all-space'
```

Environment Variables

Environment variables are not limited in length or number. Lots of applications use them, for instance, in order to set default values for configuration options.

Examples include **HOME**, **HOST**, **PATH**, and can be set as in **PATH: PATH=$HOME/bin:$PATH** for example.

Note: Putting **./** in your path is a security risk; an unfriendly user might substitute an executable which could be quite harmful. However, if you are on a single user system, you may want to violate this recommendation.

Type **env** (or **export**) to get a list of presently exported environment variables, set to get the complete set of variables.

Some variables to set (use whatever values make sense for you!):

```
EDITOR=/usr/bin/emacs
CD_PATH=$HOME:/tmp
LS_COLORS="......"
PAGER=/usr/bin/less
HISTSIZE=1000
```

An environment variable must be exported to propagate its value to a child process. You can do either of the following:

```
$ VAR=value ; export VAR
$ export VAR=value
```

You can also make one or more environment variables take effect for just one command:

```
$ LD_LIBRARY_PATH=$PWD DEBUG=3 ./foobar
```

Customizing Command Line

The default command line prompt is **$** for normal users and **#** for the root or superuser.

Customizing the command line prompt is as simple as modifying the value of the environment variable PS1. For example, to set it to display the hostname, user and current directory:

```
1  $ PS1="\h:\u:\w>"
2  c7:coop:/tmp>
```

Besides the aesthetic value of having a prettier prompt than the default value, embedding more information in the prompt can be quite useful. In the example given we have shown:

- The machine name - this is useful if you run command line windows on remote machines from your desktop; you can always tell where you are, and this can avoid many errors.
- The user name - this is particularly important if you are running as a superuser (root) and can help you avoid errors where you take what you think is a benign action and wind up crippling your system.
- The current directory - it is always important to know where you are. You certainly do not want to do something like **rm *** in the wrong directory.

Here is a table with some of the possible special characters that can be embedded in the PS1 string:

Character	Meaning	Example Output
\t	Time in HH:MM:SS	08:43:40
\d	Date in "Weekday Month Date"	Fri Mar 12
\n	Newline	
\s	Shell name	bash
\w	Current working directory	/usr/local/bin
\W	Basename of current working directory	bin
\u	User	coop
\h	Hostname	c7
\#	Command number (this session)	43
\!	History number (in history file)	1057

Note you can embed any other string you like in the prompt.

Special Characters

A number of characters have a special meaning and cause certain actions to take place. If you want to print them directly, you usually have to prefix them with a backslash (****) or enclose them in single quotes.

Redirection Special Characters

Character	Usage
\#>	Redirect output descriptor (Default **# = 1, stdout**)
<	Redirect input descriptor
>>	Append output
>&	Redirect **stdout** and **stderr** (equivalent to **.. > .. 2>&1**)

Compound Commands Special Characters

Character	Usage
|	Piping
()	Execute in a separate shell
&&	AND list
||	OR list
;	Separate commands

Expansion Special Characters

Character	Usage
{}	Lists
~	Usually means **$HOME**
$	Parameter substitution
'	Back tick; used in expression evaluation (also **$()** syntax)
$(( ))	Arithmetic substitution
[]	Wildcard expressions, and conditionals

Escapes Special Characters

Character	Usage
****	End of line, escape sequence
' '	Take exactly as is
" "	Take as is, but do parameter expansion

Other Special Characters

Character	Usage
&	Redirection and putting task in background
#	Used for comments
***?**	Used in wildcard expansion
!	Used in history expansion

Note there are three different quoting mechanisms listed above:

- **** (as in **\|**; try **echo |** vs **echo \|**)
- single quotes: preserves literal value
- double quotes: same except for **$**, **'**, and **** .

Note you can get a literal quote character by using **\'** or **\"**.

Try:

```
$ echo $HOME
$ echo \$HOME
$ echo '$HOME'
$ echo "$HOME"
```

Redirection

File descriptors:

- **0 = stdin**
- **1 = stdout**
- **2 = stderr**

less < file same as **less file** or **less 0< file**

foo > file ; redirect **stdout** (same as **foo 1> file**)

foo 2> file ; redirect **stderr**

foo >> file ; append **stdout** to **file**

foo >& file or **foo > file 2>&1**;

sends **stdout** and **stderr** to a file, but **foo >>& file** does not work ; you have to do **foo >> file 2>&1**

Note that **foo > file 2>&1** is not the same as **foo 2>&1 > file**; the order of arguments is important.

A nice non-portable trick you can use in Linux is to take advantage of the device nodes:

```
/dev/stdin
/dev/stdout
/dev/stderr
```

```
$ foo > /dev/stderr
```

Pipelines

Each step in a pipeline is a separate shell; i.e. there is a true pipeline. Be careful with redirection. Also **|&** does not work.

cat nofile | grep string produces an error if **nofile** does not exist.

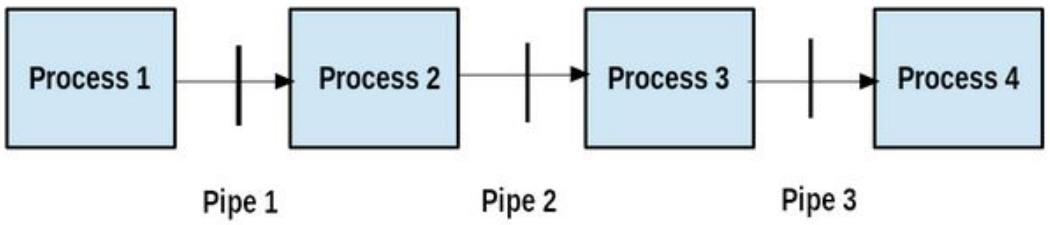
cat nofile | grep string 2>errs does not work.

cat nofile 2>&1 | grep string > errs does not work.

cat nofile 2>errors | grep string does work.

The **tee** utility can be very useful for saving output while still looking at the screen:

```
$ foobar | tee filename
$ foobar 2>&1 | tee filename
```



Command substitution and expression

There are two mechanisms for substituting the result of an operation into a command:

```
$ ls -l `which --skip-alias emacs`
$ ls -l $(which --skip-alias emacs)
```

The second form permits nesting, while the first form does not. Note that the first form has "backticks" (`) not apostrophes.

Arithmetic expressions may be evaluated in two different ways, using the **expr** utility, or the **$((..))** syntax:

For **x=3**:

Arithmetic Expression Evaluation Forms

Expression	Gives
echo $x + 1	3+1
echo $(expr $x + 1)	4
echo $((x+1))	4
echo $(($x + 1))	4
echo $(expr $x+1)	3+1

The **$((..))** syntax is more modern and preferred; **expr** is less efficient, as it invokes an external program and is trickier to use.

Note that **$var**, **$(cmd)**, **`cmd`**, and **$((...))** all expand inside double quotes.

Introduction to Filesystems

Filesystems and the FHS

- In the UNIX tradition, all filesystems and partitions are located under the root filesystem or / directory
- Other partitions are generally mounted on various subdirectory points as are network file systems which may or may not be mounted at any given time
- The **Filesystem Hierarchy Standard** (FHS), administered originally by the Free Standards Group, and now The Linux Foundation, specifies the main directories that need to be present and describes their purposes
- While most Linux distributions respect the FHS, probably none of them follow it exactly, and the last official version is usually old enough that it does not take into account some of the newest developments

Main Directories

Directory	In FHS?	Purpose
/	Yes	Primary directory of the entire filesystem hierarchy
/bin	Yes	Essential executable programs that must be available in single user mode
/boot	Yes	Files needed to boot the system, such as the kernel, initrd or initramfs images, and boot configuration files and bootloader programs
/etc	Yes	System-wide configuration files
/home	Yes	User home directories, including personal settings, files, etc.
/lib	Yes	Libraries required by executable binaries in **/bin** and **/sbin**
/lib64	No	64-bit libraries required by executable binaries in **/bin** and **/sbin**, for systems which can run both 32-bit and 64-bit programs
/media	Yes	Mount points for removable media such as CD's, DVD's, USB sticks etc.
/mnt	Yes	Temporarily mounted filesystems
/opt	Yes	Optional application software packages
/proc	Yes	Virtual pseudo-filesystem giving information about the system and processes running on it; can be used to alter system parameters
/sys	No	Virtual pseudo-filesystem giving information about the system and processes running on it; can be used to alter system parameters, is similar to a device tree and is part of the Unified Device Model
/root	Yes	Home directory for the root user
/sbin	Yes	Essential system binaries
/srv	Yes	Site-specific data served up by the system; seldom used
/tmp	Yes	Temporary files; on many distributions lost across a reboot and may be a ramdisk in memory
/usr	Yes	Multi-user applications, utilities and data; theoretically read-only
/var	Yes	Variable data that changes during system operation

A system should be able to boot and go into single user, or recovery mode, with only the **/bin**, **/sbin**, **/etc**, **/lib** and **/root** directories mounted, while the contents of the **/boot** directory are needed for the system to boot in the first place.

Many of these directories (such as **/etc** and **/lib**) will generally have subdirectories associated either with specific applications or sub-systems, with the exact layout differing somewhat by Linux distribution. Two of them, **/usr** and **/var**, are relatively standardized and worth looking at.

Directories Under /usr

Directory	Purpose
/usr/bin	Non-essential binaries and scripts, not needed for single user mode; generally this means user applications not needed to start system
/usr/include	Header files used to compile applications
/usr/lib	Libraries for programs in **/usr/bin** and **/usr/sbin**
/usr/lib64	64-bit libraries for 64-bit programs in **/usr/bin** and **/usr/sbin**
/usr/sbin	Non-essential system binaries, such as system daemons
/usr/share	Shared data used by applications, generally architecture-independent
/usr/src	Source code, usually for the Linux kernel
/usr/X11R6	X Window files; generally obsolete
/usr/local	Local data and programs specific to the host; subdirectories include **bin**, **sbin**, **lib**, **share**, **include**, etc.

Directories Under /var

Directory	Purpose
/var/ftp	Used for **ftp** server base
/var/lib	Persistent data modified by programs as they run
/var/lock	Lock files used to control simultaneous access to resources
/var/log	Log files
/var/mail	User mailboxes
/var/run	Information about the running system since the last boot
/var/spool	Tasks spooled or waiting to be processed, such as print queues
/var/tmp	Temporary files to be preserved across system reboot; sometimes linked to **/tmp**
/var/www	Root for website hierarchies

Partitions

Under Linux, disks are divided into partitions; the term slices is not often used, but when it is, it is used interchangeably with the term partitions.

Up to four primary partitions can be created and information stored about them in the MBR (Master Boot Record). More flexibility can be obtained by creating up to three primary partitions and an extended partition, which can contain as many logical partitions as can be accommodated, which may depend on the type of disk involved. For example, SCSI disks can have only up to sixteen partitions.

The Linux kernel discovers all pre-attached hard disks during system boot, and there is normally no configuration files required to inform about what is present. In hotplug situations, the udev system will find disks upon insertion in the system and read in their partition tables.

The command line utility for creating and examining hard disk partitions is **fdisk**; to see all currently attached device, you can do:

```
$ sudo /sbin/fdisk -l

Disk /dev/sda: 2000.4 GB, 2000398934016 bytes, 3907029168 sectors
Units = sectors of 1 * 512 = 512 bytes
Sector size (logical/physical): 512 bytes / 4096 bytes
I/O size (minimum/optimal): 4096 bytes / 4096 bytes
Disk label type: dos
Disk identifier: 0x000852df

  Device Boot       Start         End      Blocks    Id    System
/dev/sda1            2048  1048578047   524288000    8e    Linux LVM
/dev/sda2      1048578048  2097154047   524288000    8e    Linux LVM
/dev/sda3      2097154048  3907028991   904937472     5    Extended
/dev/sda5      2097156096  3145732095   524288000    8e    Linux LVM
/dev/sda6      3890448384  3907028991     8290304    82    Linux swap / Solaris

Disk /dev/sdb: 256.1 GB, 256060514304 bytes, 500118192 sectors
Units = sectors of 1 * 512 = 512 bytes
Sector size (logical/physical): 512 bytes / 4096 bytes
I/O size (minimum/optimal): 4096 bytes / 4096 bytes
Disk label type: dos
Disk identifier: 0x00089e7f

   Device Boot     Start        End     Blocks   Id   System
/dev/sdb1           2048   40962047   20480000   83   Linux
/dev/sdb2       40962048  500118191  229578072   83   Linux

Disk /dev/sdc: 256.1 GB, 256060514304 bytes, 500118192 sectors
Units = sectors of 1 * 512 = 512 bytes
Sector size (logical/physical): 512 bytes / 4096 bytes
I/O size (minimum/optimal): 4096 bytes / 4096 bytes
Disk label type: dos
Disk identifier: 0x00022650

   Device Boot      Start         End      Blocks   Id  System
/dev/sdc1            2048   500117503   250057728   83  Linux

Disk /dev/loop0: 2562 MB, 2562695168 bytes, 5005264 sectors
Units = sectors of 1 * 512 = 512 bytes
Sector size (logical/physical): 512 bytes / 512 bytes
I/O size (minimum/optimal): 512 bytes / 512 bytes

```

The **fdisk** utility can be used to create and remove partitions and change their type.

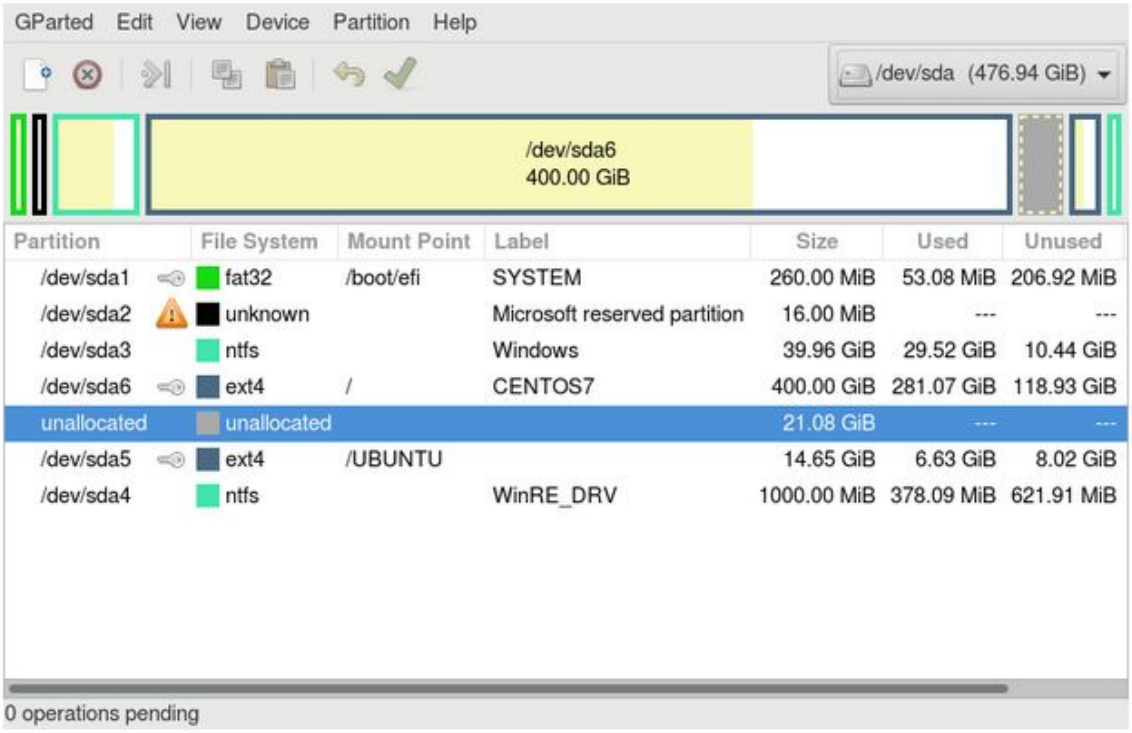
Note that **fdisk** does not allow you to move partitions or resize them. Resizing has to be done in two steps; if you are increasing, you have to increase the size of the partition, and then increase the filesystem size (for example, with **resize2fs**); if you are decreasing the size, you have to decrease the size of the filesystem and then the partition.

Partitions can be formatted for various filesystems with the **mkfs** command, or more usually, with specific commands for each type of filesystem. For example, either of the two following commands:

```
$ sudo mkfs -t ext4 /dev/sda10
$ sudo mkfs.ext4 /dev/sda10
```

will place an ext4 filesystem on **/dev/sda10** with default options.

The **gparted** utility (and some equivalents) let you do all these operations in a graphical user-friendly manner. Starting this up (as root) gives:



Partitioning Considerations

Partitioning Scheme

The exact partitioning scheme you use depends on your needs, and is quite flexible. For the most basic commonly used scheme you would have three partitions:

- **`/boot`** is relatively small, typically 100-200 MB, and holds kernels and other boot-related materials; these files are vital and rarely change, and it is safer to keep them on a separate partition
- **`/`** contains everything else and is as large as you need; depending on your distribution, system files and applications and basic programs and development tools will probably chew up 3-8 GB of space
- The **swap** partition should be at least as big as the amount of memory on your system; you can use swap files instead of partitions, but this is a weaker method due to both efficiency and stability considerations

Other Ways to Set Up Partitions

- There are many other ways to set up your partitions, especially on multi-user systems and on systems that use a lot of disk space and have large data files
- This may also depend on the kind of storage media you are using and where it is most efficient to use your fastest and slowest hardware, largest and smallest capacity devices, etc.
- Often the **`/home`** directory is put on a separate partition (or disk)
- **`/usr`** which is relatively static may be put on a separate partition, as might **`/var`** which is quite volatile, and **`/tmp`**, which is temporary
- In addition, one or more of these partitions might be available only as a network share, as in the use of NFS, and not even be mountable until the system is well-booted

Logical Volume Management (LVM)

- Use of LVM (Logical Volume Management) introduces even more flexibility, as many logical volumes can be placed on a group of physical volumes which can span more than one disk in a transparent way
- It is easy to dynamically resize and move such partitions

Paths

The path is a critical aspect of your environment, and is encapsulated in the **PATH** environment variable. On an RHEL 7 system for a user named **student**, we get:

```
$ echo $PATH
/usr/lib64/qt-3.3/bin:/usr/lib64/ccache:/usr/local/bin:/usr/bin:\
/usr/local/sbin:/usr/sbin:/home/student/.local/bin:/home/student/bin
```

(Note we have had to split the path across across two lines in the output.)

When a user tries to run a program, the path is searched (from left to right) until an executable program or script is found with that name. You can see what would be found with the **which** command, as in:

```
$ which --skip-alias emacs
/usr/bin/emacs
```

Note that if there was a **/usr/local/bin/emacs**, it would be executed instead, since it is earlier in the path.

It is easy to add directories to your path, as in:

```
$ MY_BIN_DIR=$HOME/my_bin_dir
$ export PATH=$MY_BIN_DIR:$PATH
$ export PATH=$PATH:$MY_BIN_DIR
```

with the first form prepending your new directory and the second appending it to the path.

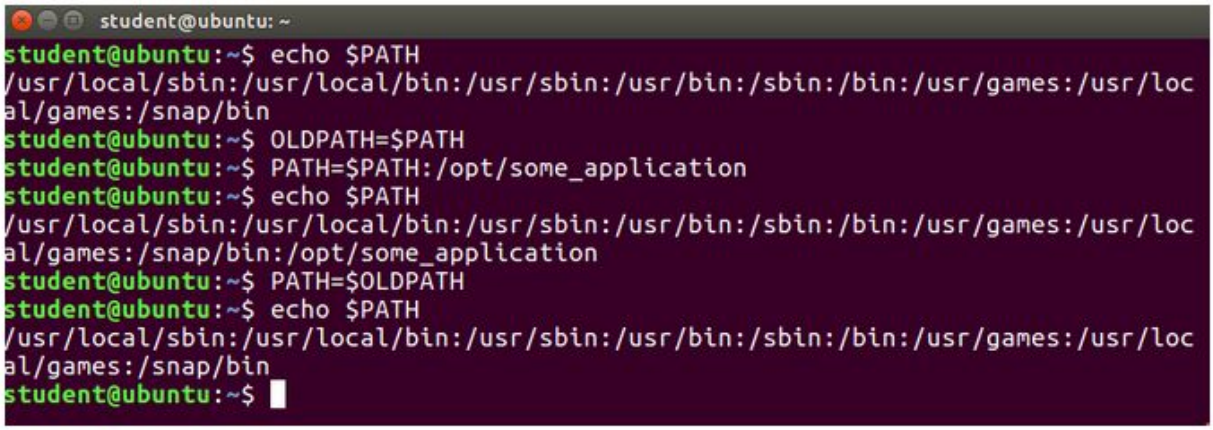
Note that the current directory is noted by **./** and the directory up one level by **../**.

The current directory is never placed in the path by default. Thus, if you want to run **foobar** in the current directory, you must say:

```
$ ./foobar
```

for it to work.

You can save changes to your path by putting them in your shell initialization file, **.bashrc** in your home directory.



Another useful path variable is **CDPATH** which is searched when you change directories. For example:

```
$ cd bin
-bash: cd: usr: No such file or directory
$ export CDPATH=/usr:$CDPATH
$ cd bin
/usr/bin
```

Hard and Symbolic (Soft) Links

The **ln** program can be used to create hard links and (with the **-s** option) soft links, also known as symbolic links or symlinks.

Suppose **file1** already exists. A hard link to it is created with the command:

```
$ ln file1 file2
$ ls -li file1 file2

84 -rw-rw-r-- 2 coop coop 1551 Jun 16 16:28 file1
84 -rw-rw-r-- 2 coop coop 1551 Jun 16 16:28 file2
```

Note that two files now appear to exist. However, a closer inspection of the file listing shows that this is not quite true.

The **-i** option to **ls** prints out in the first column the inode number, which in UNIX filesystems is unique for each file object. This field is the same for both of these files; what is really going on here is that it is only one file, but it has two names. The 2 that appears later in the **ls** listing indicates that this inode has two links to it.

Let's consider another example:

```
$ ls -li /bin/g*zip

194339 -rwxr-xr-x 3 root root 62872 Jan 14 13:06 /bin/gunzip
194339 -rwxr-xr-x 3 root root 62872 Jan 14 13:06 /bin/gzip
```

which shows that **gzip** and **gunzip** are both only one program and the executable is only one file; whether it compresses or decompresses files depends on which name it is invoked with, which is always available as **argv[0]** when the program executes.

Hard links are very useful and they save space, but you have to be careful with their use, sometimes in subtle ways. For one thing, if you remove either **file1** or **file2** in the above example, the inode object (and the remaining file name) will remain, which is generally desirable.

However, if you edit one of the files, exactly what happens depends on your editor; most editors, including vi and emacs, will retain the link by default, but it is possible that modifying one of the names may break the link and result in the creation of two objects.

Symbolic (or soft) links are created with the **-s** option, as in:

```
$ ln -s file1 file2
$ ls -li file1 file2

84 -rw-rw-r-- 1 coop coop 1551 Jun 16 16:28 file1
85 lrwxrwxrwx 1 coop coop    5 Jun 16 16:43 file2 -> file1
```

Notice **file2** no longer appears to be a regular file, and it clearly points to **file1** and has a different inode number.

Symbolic links take no extra space on the filesystem (unless their names are very long), as they are stored directly in the directory inode. They are extremely convenient, as they can easily be modified to point to different places.

Unlike hard links, soft links can point to objects even on different filesystems (or partitions), which may or not be currently mounted or even exist. In the case where the link does not point to a currently mounted or existing object, one obtains a dangling link.

The **symlinks** utility can be used to examine current symbolic links, as in:

```
c7:/usr/share/gdb/auto-load>symlinks -rv .

relative: /usr/share/gdb/auto-load/lib -> usr/lib
dangling: /usr/share/gdb/auto-load/sbin -> usr/sbin
relative: /usr/share/gdb/auto-load/lib64 -> usr/lib64
dangling: /usr/share/gdb/auto-load/bin -> usr/bin
```

easily be modified to point to different places.

System Boot

GRUB

- Virtually all x86-based physical Linux systems (outside the embedded sphere) today use **GRUB** (**GR**and **U**nified **B**ootloader) to handle the early phases of system startup
- Other platforms may have other equivalents, such as ELILO used on IA64 (Itanium), and Das U-Boot used on many embedded configurations
- Some important features of GRUB are:
  - Alternative operating systems can be chosen at boot time
  - Alternative kernels and/or initial ramdisks can be chosen at boot time for a given operating system
  - Boot parameters can be easily changed at boot time without having to edit configuration files in advance

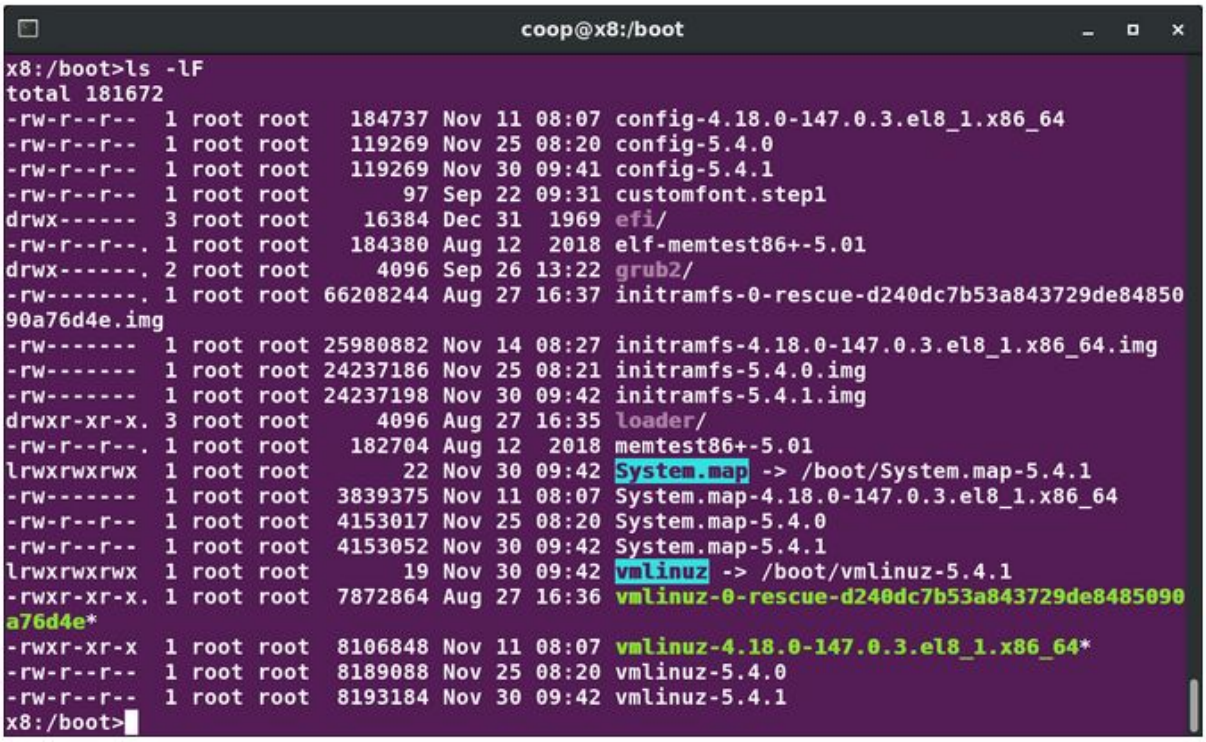
- Older Linux distributions (such as RHEL6) use older GRUB versions (1.0 or below), while newer ones are based on GRUB 2; while details are different between the versions, the basic philosophy is the same
- In GRUB 2 the basic configuration file is **`/boot/grub/grub.cfg`** or **`/boot/grub2/grub.cfg`**
- This file is auto-generated when you run **update-grub** or **grub2-mkconfig** (depending on distribution), based on files in the **`/etc/grub.d`** and **`/etc/default/grub`**; it should not be edited by hand
- On some purely EFI systems, this file might be found in a directory like **`/boot/efi/EFI/redhat/grub.cfg`**
- The essential configuration file contains some global information, and then a **stanza** for each operating system or kernel configured

Boot

- On x86 systems, boot begins with the **BIOS** identifying and initializing all system and attached peripheral devices
- If permitted by BIOS settings, the system can then boot off a peripheral device such as a CD, DVD, floppy, or USB drive, or through network PXE; more likely it will boot off the configured hard disk
- The **MBR** (Master Boot Record) contains both the **partition table** and the **boot loader**, a short program that is responsible for loading the operating system, and thus has to have at least sufficient smarts to locate and load the kernel from disk
- This kernel can have any name; usually on Linux systems it is called something like **`vmlinuz-4.18.0`** which includes the kernel version number
  - The **z** in **`vmlinuz`** indicates the kernel is compressed; it self-decompresses upon loading
  - In most circumstances it is placed in the **`/boot`** directory which is often on its own partition

Reading /boot directory

A **/boot** directory might look like the one in the screenshot below:



In this example, there are multiple possible kernels to boot into, each of which has four files associated with it:

- **vmlinuz** is the compressed kernel
- **initramfs** contains a complete initial root filesystem which is loaded as a ramdisk, as well as some essential kernel modules (generally, device drivers) and the programs needed to load them, that are required to load the real filesystem, at which point it is discarded
- **config** contains all the details about how the kernel was compiled; it is not needed for system operation
- **System.map** lists the complete kernel symbol table; it is used only for debugging purposes

Depending on which Linux distribution is being used, there will be variations in the above possible kernels. For example, the **initramfs** file may actually be called **initrd**.

System Initialization

init

- **`/sbin/init`** (usually just called **init**) is the first user-level process (or task) that runs on the system and continues to run until the system is shutdown
- Traditionally it has been considered the parent of all user processes, although technically that is not true as some processes are started directly by the kernel
- init coordinates the later stages of the boot process, configures all aspects of the environment and starts the processes needed for logging into the system; it also works closely with the kernel in cleaning up after processes when they terminate

SysVinit

- Traditionally, nearly all distributions based the init process on UNIX's venerable **SysVinit**
- This scheme was developed decades ago under rather different circumstances:
  - The target was multi-user mainframe systems (not personal computers, laptops, and other devices)
  - The target was a single processor system
  - Startup (and shutdown) time was not an important matter; it was far less important than getting things right
    - Startup was viewed as a serial process, divided into a series of sequential stages; each stage required completion before the next could proceed, so it didn't easily take advantage of the parallel processing that could be done on multiple processors or cores
    - Shutdown/reboot was seen as a relatively rare event and exactly how long it took was not considered important

New Methods of Controlling System Startup

To deal with these intrinsic limitations in SysVinit, new methods of controlling system startup were developed and have replaced it in all new systems. Two main schemes were adopted by Linux distributors:

- Upstart
  - Developed by Ubuntu and first included in the 6.10 release in 2006, and made the default in the 9.10 release in 2009
  - Adopted in Fedora 9 (in 2008) and in RHEL 6 and its clones, such as CentOS, Scientific Linux and Oracle Linux, and in openSUSE it was offered as an option since version 11.3
  - It has also been used in various embedded and mobile devices
- **systemd** is more recent and Fedora was the first major distribution to adopt it in 2011

systemd

- RHEL 7 is based on **systemd** and every other major Linux distribution has adopted it and has made it the default or announced plans to do so; even Ubuntu phased out Upstart and is now systemd-based
- We will discuss systemd commands, in particular the use of **systemctl**, when we discuss controlling system services; starting and stopping, enabling and disabling at boot, showing status, etc.
- Migration to systemd was non-trivial and missing features can be very disabling, so there are essential required compatibility layers
- Wrappers will ensure one can use SysVinit utilities and methods for quite some time, even if under the hood things are quite different
- However, eventually they will atrophy and disappear, so learning systemd is the best investment of your learning efforts

System Runlevels

As a SysVinit system starts, is passes through a sequence of runlevels which define different system states; they are numbered from 0 to 6:

- Runlevel 0 is reserved for the **system halt** state
- Runlevel 1 for **single-user mode**
- Runlevel 6 for **system reboot**
- The other runlevels are used to define what services are running for a normal system and different distributions define them somewhat differently
    - Example: On Red Hat-based systems, runlevel 2 is defined as a running system without networking or X, runlevel 3 includes networking, and runlevel 5 includes networking and X

System Runlevels

Here is a table summarizing the levels:

Runlevel	Meaning
S, s	Same as 1
0	Shutdown system and turn power off
1	Single user mode
2	Multiple user, no NFS, only text login
3	Multiple user, with NFS and network, only text login
4	Not used
5	Multiple user, with NFS and network, graphical login with X
6	Reboot

The current runlevel can be simply displayed with the **runlevel** command, as in:

```
$ runlevel
NS
```

where the first character is the previous level; **N** means unknown.

telinit can be used to change the runlevel of the system. For example, to go from runlevel 3 to runlevel 5, type:

```
$ sudo /sbin/telinit 5
```

When the init process is started, the first thing it does is to read **/etc/inittab**. Historically, this file told init which scripts to run to bring the system up each runlevel, and was done with a series of lines, one for each runlevel:

```
id:runlevel(s):action:process
```

where:

- **id:** a unique 1-4 character identification for the entry
- **runlevel(s):** zero or more single character or digit identifiers which identify which runlevel the action will be taken for
- **action:** describes the action to be taken
- **process:** specifies the process to be executed.

However, in more recent systems, the only un-commented line and the only thing being set in this file is the default runlevel with the line:

```
id:5:initdefault
```

This is the level to stop at when booting the system. However, if another value is specified on the kernel command line, init ignores the default. This is done by simply appending the right integer to the kernel command line. The default level is usually 5 for a full multi-user, networked graphical system, or 3 for a server without a graphical interface.

systemd-based systems do not use **/etc/inittab** at all, and just contain a file with no uncommented content, so as to not break outdated scripts. However, some distributions still maintain the notion of runlevels, which are defined in terms of systemd targets, and so you can use commands like the **telinit** one described earlier.

From within a graphical terminal (**gnome-terminal**, **konsole**, etc.), we kill the current graphical desktop.

Your method will depend on your distribution, your greeter program (**gdm**, **lightdm**, **kdm**), and whether you have a SysVinit or systemd system.

First, we will bring down the GUI, which, depending on your system, will be done with one of the following commands:

```
$ sudo systemctl stop gdm
$ sudo telinit 3
```

Now, we restart the GUI from the text console with one of the following commands:

```
$ sudo systemctl start gdm
$ sudo telinit 5
```

Memory

Linux uses a virtual memory system (VM), as do all modern operating systems: the virtual memory is larger than the physical memory.

Each process has its own, protected address space. Addresses are virtual and must be translated to and from physical addresses by the kernel whenever a process needs to access memory.

The kernel itself also uses virtual addresses; however the translation can be as simple as an offset depending on the architecture and the type of memory being used.

The kernel allows fair shares of memory to be allocated to every running process, and coordinates when memory is shared among processes. In addition, mapping can be used to link a file directly to a process's virtual address space. Furthermore, certain areas of memory can be be protected against writing and/or code execution.

The **free** utility gives a very terse report on free and used memory in your system:

```
$ free -mt
              total        used        free      shared  buff/cache   available
Mem:          15893        3363         175         788       12354       11399
Swap:          8095           0        8095
Total:        23989        3363        8271
```

where the options cause the output to be expressed in MB's. See **man free** to see possible options.

This system has 16 GB of RAM and a 8 GB swap partition. At the moment, this snapshot was taken the system was pretty inactive and not doing all that much. Yet, the amount of memory being used is appreciable (if you include the memory assigned to the cache).

However, a lot of the memory being used is in the page cache, most of which is being used to cache the contents of files that have recently been accessed. If this cache is released, the memory usage will decrease significantly. This can be done by doing (as root user):

```
$ sudo su
# echo 3 > /proc/sys/vm/drop_caches
# exit
$ free -mt
              total        used        free      shared  buff/cache   available
Mem:          15893        3370       11103         788        1419       11419
Swap:          8095           0        8095
Total:        23989        3370       19199
```

If we had only wanted to drop the page cache, we would have echoed a 1, not a 3; we have also dropped the dentry and inode caches, which is why the freed memory is more than that released from the page cache.

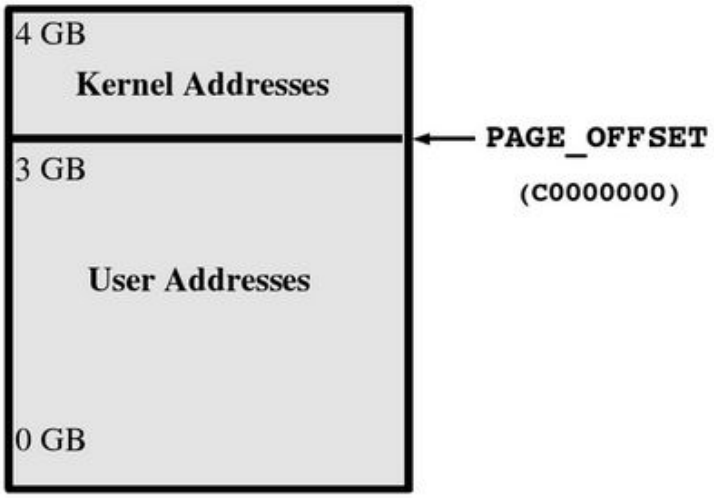
A more detailed look can be obtained by looking at **/proc/meminfo**:

```
$ cat /proc/meminfo

MemTotal:       16275064 kB
MemFree:        11059060 kB
MemAvailable:   11525932 kB
Buffers:           30416 kB
Cached:          1598188 kB
SwapCached:            0 kB
Active:          3880768 kB
Inactive:        1105144 kB
Active(anon):    3295948 kB
Inactive(anon):   994524 kB
Active(file):     584820 kB
Inactive(file):   110620 kB
Unevictable:        3596 kB
```

The output will depend somewhat on kernel version, and you should not write scripts that overly depend on certain fields being in this file.

In the following diagram (for 32-bit platforms), the first 3 GB of virtual addresses are used for user-space memory and the upper GB is used for kernel-space memory. Other architectures have the same setup, but differing values for **PAGE_OFFSET**; for 64-bit platforms, the value is in the stratosphere.



While Linux permits up to 64 GB of memory to be used on 32-bit systems, the limit per process is a little less than 3 GB. This is because there is only 4 GB of address space (i.e. it is 32-bit limited) and the topmost GB is reserved for kernel addresses. The little is somewhat less than 3 GB because of some address space being reserved for memory-mapped devices.

It is important to remember that applications do not write directly to storage media such as disks; they interface with the virtual memory system and data blocks written are generally first placed into cache or buffers, and then are flushed to disk when it is either convenient or necessary. Thus, in most systems, more memory is used in this buffering/caching layer than for direct use by applications for other purposes.

For a comprehensive review, see Ulrich Drepper's article "What Every Programmer Should Know About Memory". This covers many issues in depth, such as proper use of cache, alignment, NUMA, virtualization, etc.

Managing the memory on 32-bit machines with large amounts of memory (especially over 4 GB) is far more complex than it is in 64-bit systems.

It is hard to think of a good reason to be acquiring purely 32-bit hardware anymore for use as heavy iron; there is still plenty of use for 32-bit systems in the embedded world, etc., but there memory is not expected to be large enough to complicate memory management.

Of course, you may be running 32-bit applications on 64-bit hardware, and that may lead occasionally to questions which may be subtle.

However, to keep things simple, we will not focus on 32-bit systems with a lot of memory, as they are more and more becoming dinosaurs.

Swap

Linux employs a virtual memory system in which the operating system can function as if it had more memory than it really does. This kind of memory overcommission functions in two ways:

- Many programs do not actually use all the memory they are given permission to use. Sometimes, this is because child processes inherit a copy of the parent's memory regions utilizing a COW (Copy On Write) technique, in which the child only obtains a unique copy (on a page-by-page basis) when there is a change.
- When memory pressure becomes important, less active memory regions may be swapped out to disk, to be recalled only when needed again.

Such swapping is usually done to one or more dedicated partitions or files; Linux permits multiple swap areas, so the needs can be adjusted dynamically. Each area has a priority and lower priority areas are not used until higher priority areas are filled.

In most situations, the recommended swap size is the total RAM on the system. You can see what your system is currently using for swap areas with:

```
$ cat /proc/swaps
Filename                Type            Size    Used    Priority
/dev/sda9               partition       4193776 0       -1
/dev/sdb6               partition       4642052 0       -2
```

and current usage with:

```
$ free
              total        used        free      shared      buffers       cached
Mem:       4047236     3195080      852156           0      818480     1430940
Swap:      8835828           0     8835828
```

The only commands involving swap are **mkswap** for formatting a swap file or partition, **swapon** for enabling one (or all) swap area, and **swapoff** for disabling one (or all) swap area.

At any given time, most memory is in use for caching file contents to prevent actually going to the disk any more than necessary, or in a sub-optimal order or timing. Such pages of memory are never swapped out as the backing store is the files themselves, so writing out to swap would be pointless; instead, dirty pages (memory containing updated file contents that no longer reflect the stored data) are flushed out to disk.

It is also worth pointing out that Linux memory used by the kernel itself, as opposed to application memory, is never swapped out, in distinction to some other operating systems.

Threading Models

Processes and Threads

- A **process** is a running instance of a program and contains information about environment variables, file descriptors and current directory, etc.
- It can contain one or more **threads**, each of which has the same process ID and shares the same environment, memory regions (except for stack), etc.
- While Linux permits thread (or process) creation through the low level **`clone()`** system call, the most usual method is to use **`pthread_create()`** which is part of the standard **POSIX Threads** (**pthreads**) library

Implementation of pthreads

- In a perfect world any pthreads compliant program should be *write once, use anywhere*
- The real world is not always so simple; for example, one operating system's implementation of pthreads may be more forgiving of errors than another and when an application is ported it may stop working
- Instances have been seen, where **Solaris** was automatically initializing improperly uninitialized mutexes, while Linux did not and thus led to program failure
- Also system differences such as default stack sizes can lead to unforeseen problems
- Other operating systems may also introduce their own specific multi-threading environments, such as Solaris threads
- Porting applications from such environments may be simple or difficult depending on APIs; it is best to stick with pthreads if possible, for portability alone

Networking

The vast majority of network programming in Linux is done using the socket interface. Thus, standards-compliant programs should require little massage to work properly with Linux.

Note, however, there are many enhancements and new features in the Linux networking implementation, such as new kinds of address and protocol families. For example, Linux offers the **netlink** interface, which permits opening up socket connections between kernel sub-systems and applications (or other kernel sub-systems). This has been effectively deployed to implement firewall and routing applications.

Historically, the wired Ethernet network devices have been known by a name such as **eth0**, **eth1**, etc., while wireless devices have had names like **wlan0**, **wlan1**, etc.

Basic information about active network interfaces on your system is gathered through the **ifconfig** utility:

```
$ /sbin/ifconfig
eth0      Link encap:Ethernet  HWaddr 00:22:15:2B:64:A6
          inet addr:192.168.1.100  Bcast:192.168.1.255  Mask:255.255.255.0
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:163529 errors:0 dropped:0 overruns:0 frame:0
          TX packets:112693 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:183642176 (175.1 MiB)  TX bytes:12101864 (11.5 MiB)
          Interrupt:18
eth1      Link encap:Ethernet  HWaddr 00:22:15:2B:63:BE
          inet addr:192.168.0.101  Bcast:192.168.0.255  Mask:255.255.255.0
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:162597 errors:0 dropped:0 overruns:0 frame:0
          TX packets:56710 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:206698846 (197.1 MiB)  TX bytes:75532637 (72.0 MiB)
```

Information displayed includes information about the hardware MAC address, the MTU (maximum transfer unit), and the IRQ the device is tied to. Also displayed are the number of packets and bytes transmitted, received, or resulting in errors.

This machine has two network cards bound to **eth0** and **eth1**, and the loopback interface, **lo**, which handles network traffic bound to the machine. Note you can see the statistical information in abbreviated form by looking at **/proc/net/dev**, and in one quantity per line display in **/sys/class/net/eth0/statistics**:

```
$ ls -l /sys/class/net/eth0/statistics
total 0
-r--r--r-- 1 root root 4096 Mar 26 17:21 collisions
-r--r--r-- 1 root root 4096 Mar 26 17:30 multicast
-r--r--r-- 1 root root 4096 Mar 26 17:20 rx_bytes
-r--r--r-- 1 root root 4096 Mar 26 17:30 rx_compressed
-r--r--r-- 1 root root 4096 Mar 26 17:30 rx_crc_errors
-r--r--r-- 1 root root 4096 Mar 26 17:30 rx_dropped
-r--r--r-- 1 root root 4096 Mar 26 17:20 rx_errors
-r--r--r-- 1 root root 4096 Mar 26 17:30 rx_fifo_errors
-r--r--r-- 1 root root 4096 Mar 26 17:30 rx_frame_errors
-r--r--r-- 1 root root 4096 Mar 26 17:30 rx_length_errors
-r--r--r-- 1 root root 4096 Mar 26 17:30 rx_missed_errors
```

Using Predictable Network Interface Device Names

Naming Networks

- Simply naming network devices as eth0, eth1, etc. can be problematic when multiple interfaces exist, or when the order in which the system probes for and then finds them is not deterministic and can depend on kernel version and options
- Many system administrators have solved this problem in a simple manner, by hard-coding associations between hardware (MAC) addresses and device names in system configuration files and startup scripts
- A more modern approach is offered by the Predictable Network Interface Device Names scheme, which is strongly correlated with the use of udev and integration with systemd

Five Types of Names

- There are now 5 types of names that devices can be given:
  - Incorporating Firmware or BIOS provided index numbers for on-board devices, e.g. **eno1**
  - Incorporating Firmware or BIOS provided PCI Express hotplug slot index numbers, e.g. **ens1**
  - Incorporating physical and/or geographical location of the hardware connection, e.g. **enp2s0**
  - Incorporating the MAC address, e.g. **enx7837d1ea46da**
  - Using the old classic method, e.g. **eth0**
- On a machine with two onboard PCI network interfaces that would have been **eth0** and **eth1** in the older naming system:

```
$ ifconfig | grep enp
enp2s0: flags=4163<UP,BROADCAST,RUNNING,MULTICAST> mtu 1500
enp4s2: flags=4099<UP,BROADCAST,MULTICAST> mtu 1500
```

- It is easy to turn off the new scheme and go back to the classic names, if so desired

To bring a network connection up and assign a static address, you can do:

```
$ sudo /sbin/ifconfig eth0 up 192.168.1.100
```

To bring it up and get it an assigned address from a DHCP server, you can do:

```
$ sudo /sbin/ifconfig eth0 up
$ sudo /sbin/dhclient eth0
```

While **ifconfig** has been used reliably for many years, the **ip** utility is newer (and far more versatile). On a technical level, it is more efficient because it uses **netlink** sockets, rather than **ioctl** system calls.

ip can be used for a wide variety of tasks. It can be used to display and control devices, routing, policy-based routing, and tunneling. The basic syntax is:

```
ip [ OPTIONS ] OBJECT { COMMAND | help }
```

Some examples:

- Show information for all network interfaces:

```
$ ip link
```

- Show information for the **eth0** network interface:

```
$ ip -s link show eth0
```

- Set the IP address for **eth0**:

```
$ sudo ip addr add 192.168.1.7 dev eth0
```

- Bring **eth0** down:

```
$ sudo ip link set eth0 down
```

- Set the MTU to 1480 bytes for **eth0**:

```
$ sudo ip link set eth0 mtu 1480
```

- Set the networking route:

```
$ sudo ip route add 172.16.1.0/24 via 192.168.1.5
```

Basic Commands and Utilities

Many basic commands and utilities are the same in Linux and other UNIX-like operating systems. While there may be some variation in some of the options and syntax, the purpose remains the same. Here are lists of these commands grouped by general area of coverage:

File Compression

bunzip2, **bzcat**, **bdiff**, **bzip2**, **bzless**

gunzip, **gzexe**, **gzip**, **zcat**, **zless**

zip, **upzip**

xz, **unxz**, **xzcat**

File Ownership, Permissions and Attributes

attr, **chgrp**, **chown**, **chmod**

Files

awk, **basename**, **cat**, **col**, **cp**, **cpio**, **csplit**, **cut**, **dd**, **diff**, **dirname**, **egrep**, **expand**, **file**, **fgrep**, **fmt**, **grep**, **head**, **join**, **less**, **more**, **sed**, **tail**, **tar**

Filesystem

cd, **chroot**, **df**, **dirs**, **du**, **fdisk**, **fsck**, **fuser**, **ln**, **ls**, **mkdir**, **mv**, **pushd**, **popd**, **rm**, **rmdir**

Networking

arp, **domainname**, **finger**, **ftp**, **host**, **hostname**, **ip**, **route**, **ifconfig**, **netstat**

Job Control

at, **atrm**, **batch**, **crontab**, **exec**, **exit**, **ipcs**, **ipcrm**, **kill**, **killall**

Expression Evaluation

bc, **dc**, **eval**, **expr**, **factor**, **false**, **true**

There are many other commands and utilities that could be added to this list.

File Transfer Tools

File Transfer Tools

- On Linux systems file transfer tools are plentiful and familiar
- **FTP** clients abound, from the familiar command line ftp and enhanced (or dumbed down depending on your point of view) versions such as lftp, ncftp and qftp
  - An excellent graphical client is provided by **FileZilla**, and of course any competent browser can handle FTP downloads, although not uploads
  - The dominant ftp server in use on Linux systems is **vsftpd**, which is both simple and highly configurable
- Somewhat more general transfer opportunities are provided by both **curl** and **wget** and their associates, which can handle protocols such as ftp, http, etc.

- Secure Shell clients and servers are always available (**ssh** and **sshd**) using the **OpenSSH** implementation
  - These include both scp and sftp utilities
  - Note that many organizations do not permit plain FTP because passwords may be transmitted in plain text
- The **rsync** server and client utilities permit very rapid file transfer and synchronization between remote computers, and use secure encryption methods as well
- Old fashioned insecure programs such as rsh, rcp, etc. may be available, but are strongly discouraged

Monitoring and Performance Utilities

Linux distributions come with many standard performance and profiling tools already installed. Some of them may be familiar from other UNIX-like operating systems, while others were developed specifically for Linux.

Many of these tools gather their information from the **/proc** pseudo-filesystem. There are also graphical system monitors that, while hiding many of the details, are still extremely useful. We will consider available graphical interfaces after detailing the command line utilities.

Before considering the main utilities in some detail, we will give a brief summary. We will break them down by type, although some of the utilities have overlapping domains of coverage. We will also give the name of the package they belong to, which is not important and may vary among different Linux distributions and releases.

Process and Load Monitoring Utilities

Utility	Purpose	Package
top	Process activity, dynamically updated	**procps**
uptime	How long the system is running and the average load	**procps**
ps	Detailed information about processes	**procps**
pstree	A tree of processes and their connections	**psmisc** (or **pstree**)
mpstat	Multiple processor usage	**sysstat**
iostat	CPU utilization and I/O statistics	**sysstat**
sar	Display and collect information about system activity	**sysstat**
numastat	Information about NUMA (Non-Uniform Memory Architecture)	**numactl**
strace	Information about all system calls a process makes	**strace**

Memory Monitoring Utilities

Utility	Purpose	Package
free	Brief summary of memory usage	**procps**
vmstat	Detailed virtual memory statistics and block I/O, dynamically updated	**procps**
pmap	Process memory map	**procps**

I/O Monitoring Utilities

Utility	Purpose	Package
iostat	CPU utilization and I/O statistics	**sysstat**
iotop	I/O statistics including per process	**iotop**
sar	Display and collect information about system activity	**sysstat**
vmstat	Detailed virtual memory statistics and block I/O, dynamically updated	**procps**

Network Monitoring Utilities

Utility	Purpose	Package
netstat	Detailed networking statistics	**netstat**
iptraf	Gather information on network interfaces	**iptraf**
tcpdump	Detailed analysis of network packets and traffic	**tcpdump**
wireshark	Detailed network traffic analysis	**wireshark**

Graphical Monitoring Tools

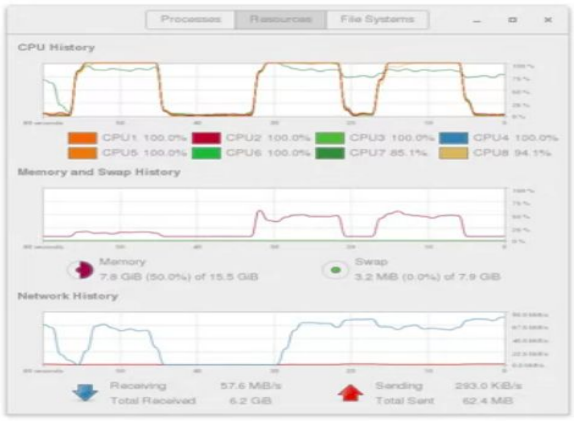
gnome-system-monitor

- gnome-system-monitor is a simple graphical monitoring tool; it is installed on any Linux distribution that provides the GNOME desktop manager (even if you are running another desktop manager, such as KDE, it will be available)
- All statistics are generated in real time, and it is easy to examine CPU load, memory and swap usage, and network bytes transmitted and received
- However, it does not have facilities to save data and the choice of display items is limited

ksysguard

- **ksysguard** (KDE System Guard) is provided on any Linux distribution that supplies the KDE desktop manager (if you are running another desktop manager it will be available)
  - On Ubuntu systems you need to install the **ksysguard** package
  - On Red Hat-based systems you need to install both the **ksysguard** and **ksysguardd** packages
- It has far more extensive capabilities than gnome-system-monitor including:
  - The ability to choose which sensors to display, by dragging and dropping them into the workspace
  - The ability to choose how many windows to display in the workspace, adjusting the numbers of rows and columns
  - The ability to create multiple worksheets and store them for future situations
  - The ability to monitor other systems besides the graphical workstation it is running on; this can be done most easily through the ssh protocol, which will launch the **ksysguardd** daemon service on the remote machine

gnome-system-monitor (Cont.)



gnome-system-monitor in action on a RHEL 7 system

ksysguard (Cont.)

ksysguard in action on a RHEL 7 system

Loading/Unloading Kernel Modules

Many facilities in the Linux kernel can either be built-in to the kernel when it is initially loaded, or dynamically added (or removed) later as modules, upon need or demand. Indeed, all but some of the most central kernel components are designed to be modular.

Such modules may or may not be device drivers; for example, they may implement a certain network protocol or filesystem. Even in cases where the functionality will virtually always be needed, incorporation of the ability to load and unload as a module facilitates development, as kernel reboots are not required to test changes.

Even with the widespread usage of kernel modules, Linux retains a monolithic kernel architecture, rather than a microkernel one. This is because once a module is loaded, it becomes a fully functional part of the kernel, with few restrictions. It communicates with all kernel sub-systems primarily through shared resources, such as memory and locks, rather than through message passing as might a microkernel.

Linux is hardly the only operating system to use modules; certainly Solaris does it, as well as does AIX, which terms them kernel extensions. However, Linux uses them in a particularly robust fashion.

Module loading and unloading must be done as the root user. If you know the full path name, you can always load the module directly with:

```
$ sudo /sbin/insmod <pathto>/module_name.ko
```

A kernel module always has a file extension of **.ko**, as in **e1000e.ko**, **ext4.ko**, or **nouveau.ko**.

Many modules can be loaded while specifying parameter values, such as in:

```
$ sudo /sbin/insmod <pathto>/module_name.ko irq=12 debug=3
```

While the module is loaded, you can always see its status with the **lsmod** command:

```
$ lsmod

Module                  Size  Used by
coretemp               16384  0
e1000e                237568  0
ptp                    20480  1 e1000e
pps_core               20480  1 ptp
```

Direct removal can always be done with:

```
$ sudo /sbin/rmmod module_name
```

Note that it is not necessary to supply the full path name or the **.ko** extension when removing a module.

From within a graphical terminal (**gnome-terminal**, **konsole**, etc.), we kill the current graphical desktop.

Your method will depend on your distribution, your greeter program (**gdm**, **lightdm**, **kdm**), and whether you have a SysVinit or systemd system.

First, we will bring down the GUI, which, depending on your system, will be done with one of the following commands:

```
$ sudo systemctl stop gdm
$ sudo telinit 3
```

Now, we restart the GUI from the text console with one of the following commands:

```
$ sudo systemctl start gdm
$ sudo telinit 5
```

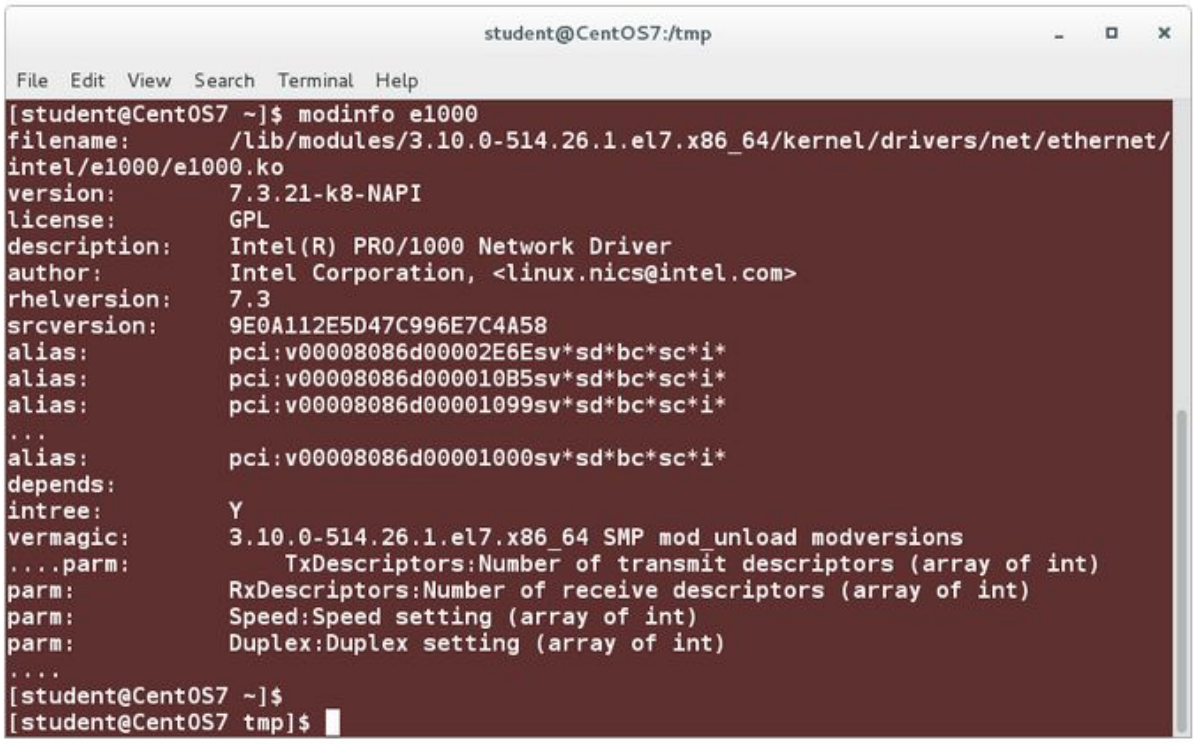
There are some important things to keep in mind when loading and unloading modules:

- It is impossible to unload a module that is being used by another module, which can be seen from the **lsmod** listing.
- It is impossible to unload a module that is being used by one or more processes, which can also be seen from the **lsmod** listing. However, there are modules which do not always keep track of this reference count, such as network device driver modules, as it would make it too difficult to temporarily replace a module without shutting down and restarting much of the whole network stack.
- When a module is loaded with **modprobe**, the system will automatically load any other modules that are required to be loaded first.
- When a module is unloaded with **modprobe -r**, the system will automatically unload any other modules being used by the module, if they are not being simultaneously used by any other loaded modules.
- Files in the **/etc/modprobe.d** directory control some parameters that come into play when loading with **modprobe**. These parameters include module name aliases and automatically supplied options. This directory also contains information about blacklisted modules, which should never be located and loaded.

The **modinfo** command can be used to find out information about kernel modules, whether they are loaded or not, as in:

```
1  $ /sbin/modinfo my_module
2  $ /sbin/modinfo <pathto>/my_module.ko
```

An example can be seen in the screenshot below.



Device Management

Types of Devices

There are three main types of devices:

- **Character devices** are sequential streams; they mainly implement **open**, **close**, **read**, and **write** functions, e.g. serial and parallel ports (**`/dev/ttyS0`**, **`/dev/lp1`**), sound cards (**`/dev/dsp0`**), etc.
- **Block devices** are randomly accessed only in block-size multiples, and I/O operations are usually cached and deal with mounted filesystems, e.g. hard drive partitions (**`/dev/sda1`**, **`/dev/sdb8`**), CD-ROMs, etc.
- **Network devices** transfer packets of data, not blocks or streams, and usually employ a socket interface; packet reception/transmission functions replace read/write operations, and there are no corresponding filesystem nodes; instead, the interfaces are identified by a name, such as **`eth0`** or **`wlan0`**.

Other Types of Devices

- There are also other types of devices which are classified somewhat differently, according to the type of controller bus they are attached to, such as **SCSI** (**S**mall **C**omputers **S**ystems **I**nterconnect) and **USB** (**U**niversal **S**erial **B**us)
- These devices share an underlying protocol regardless of function
- Besides the driver for the device itself, hard work goes into writing the driver for the controller hardware which may run many devices
- One also has **user-space drivers**, which work completely in user-space, but are given hardware access privileges; these are not as efficient as in-kernel drivers, but are less likely to bring a system down

Device Nodes

- Character and block devices have filesystem entries associated with them
- These **nodes** can be used by user-level programs to communicate with the device, using normal I/O system calls such as **`open()`**, **`close()`**, **`read()`**, and **`write()`**
- A device driver can manage more than one device node

/dev Directory

- Device nodes are normally placed in the **/dev** directory and can be created with:

  $ sudo mknod [-m mode] /dev/name <type> <major> <minor>

  e.g.

  $ mknod -m 666 /dev/mycdrv c 254 1

  or from the **mknod()** system call
- The major number identifies the driver associated with the device; all device nodes of the same type (block or character) with the same major number use the same driver
- The minor number is used only by the device driver to differentiate between the different devices it may control
  - Either different instances of the same kind of device, (such as the first and second sound card, or hard disk partition) or
  - Different modes of operation of a given device (e.g. different density floppy drive media)

/dev Directory (Cont.)

- Device numbers have meaning in user-space as well
- Two POSIX system calls, **mknod()** and **stat()** return information about major and minor numbers

```
File  Edit  View  Search  Terminal  Help
c7:/tmp>ls -l /dev
total 0
....
crw-------   1 root root       5,   1 May 26 07:05 console
....
lrwxrwxrwx   1 root root           13 May 26 07:04 fd -> /proc/self/fd
....
brw-rw----   1 root disk       7,   0 May 26 07:05 loop0
crw-rw----   1 root disk      10, 237 May 26 07:05 loop-control
.....
crw-rw----   1 root lp         6,   0 May 26 07:05 lp0
crw-rw----   1 root lp         6,   1 May 26 07:05 lp1
....
brw-rw----   1 root disk       8,   0 May 26 07:05 sda
brw-rw----   1 root disk       8,   1 May 26 07:05 sda1
brw-rw----   1 root disk       8,   2 May 26 07:05 sda2
brw-rw----   1 root disk       8,   3 May 26 07:05 sda3
....
lrwxrwxrwx   1 root root           15 May 26 07:04 stderr -> /proc/self/fd/2
lrwxrwxrwx   1 root root           15 May 26 07:04 stdin -> /proc/self/fd/0
lrwxrwxrwx   1 root root           15 May 26 07:04 stdout -> /proc/self/fd/1
crw-rw-rw-   1 root tty        5,   0 May 26 07:05 tty
crw--w----   1 root tty        4,   0 May 26 07:05 tty0
....
c7:/tmp>
c7:/tmp>
```

/dev Directory

Managing Device Nodes

- The methods of managing device nodes became clumsy and difficult as Linux evolved
- The number of device nodes lying in **/dev** and its subdirectories reached numbers in the 15,000 - 20,000 range in most installations during the 2.4 kernel series
- Nodes for all kinds of devices which would never be used on most installations were still created by default, as distributors could never be sure exactly which hardware would be needed
- Many developers and system administrators trimmed the list to what was actually needed, especially in embedded configurations, but this was essentially a manual and potentially error-prone task

- Note that while device nodes are not normal files and do not take up significant space on the filesystem, having huge directories slowed down access to device nodes, especially upon first usage
- Furthermore, exhaustion of available major and minor numbers required a more modern and dynamic approach to the creation and maintenance of device nodes
- Ideally, one would like to register devices by name; however, major and minor numbers cannot be gotten rid of altogether, as **POSIX** requires them

udev

- The **udev** method creates device nodes on the fly as they are needed; there is no need to maintain a ton of device nodes that will never be used
- The **u** in udev stands for user, and indicates that most of the work of creating, removing, and modifying devices nodes is done in user-space
- udev handles the dynamical generation of device nodes and it evolved to replace earlier mechanisms such as devfs and hotplug
- It supports persistent device naming; names need not depend on the order of device connection or plugging in - such behavior is controlled by specification of udev rules
- udev runs as a **daemon** and monitors a **netlink** socket
    - When new devices are initialized or removed the uevent kernel facility sends a message through the socket which udev receives and takes appropriate action to create or remove device nodes of the right names and properties according to the rules

udev Components

The three components of udev are:

- The **libudev** library which allows access to information about the devices
- The **udevd** daemon that manages the **`/dev`** directory
- The **udevadm** utility for control and diagnostics

Creating and Removing Device Nodes

- As devices are added or removed from the system, working with the hotplug subsystem, udev acts upon notification of events to create and remove device nodes
- The information necessary to create them with the right names, major and minor numbers, permissions, etc., are gathered by examination of information already registered in the **sysfs** pseudo-filesystem (mounted at **`/sys`**) and a set of configuration files
- The main configuration file is **`/etc/udev/udev.conf`**; it contains information such as where to place device nodes, default permissions and ownership, etc.
- By default, rules for device naming are located in the **`/etc/udev/rules.d`** directory
- By reading the **man** page for udev, one can get a lot of specific information about how to set up rules for common situations

Managing System Services

Every operating system has services which are usually started on system initialization and often remain running until shutdown. Such services may be started, stopped, or restarted at any time, generally requiring root privilege.

All relatively new Linux distributions have adopted the systemd method, which does most of the work with the **systemctl** utility.

Most older distributions, such as RHEL 6, use the **service** and **chkconfig** utilities. While older Debian-based systems use ***rc-**** programs, they also have versions of **service** and/or **chkconfig** available for install.

Generally speaking, systemd-based systems maintain backwards compatibility wrappers so one can use the older commands.

For this reason, we will only discuss in detail the systemd methods.

For an excellent summary of how to go from SysVinit to systemd, see the SysVinit to systemd Cheatsheet.

With systemd, all service management is done with the **systemctl** utility. Its basic syntax is:

```
1 $ systemctl [options] command [name]
```

We will provide some examples next.

To show the status of everything systemd controls, do:

```
1 $ systemctl
```

Show all available services:

```
$ systemctl list-units -t service --all
```

Show only active services:

```
$ systemctl list-units -t service
```

To start (activate) one or more units:

```
$ sudo systemctl start foo
$ sudo systemctl start foo.service
$ sudo systemctl start /path/to/foo.service
```

where a unit can be a service or a socket.

To stop (deactivate):

```
$ sudo systemctl stop foo.service
```

These commands are equivalent to **sudo service foo start|stop** .

Enable/disable a service:

```
$ sudo systemctl enable sshd.service
$ sudo systemctl disable sshd.service
```

This is the equivalent of **chkconfig on|off** and does not actually start the service.

Note that some **systemctl** commands in the above examples can be run as non-root user, others require running as root or with **sudo**. Furthermore, in most cases, you can omit the **.service** from the service name.

Software Management and Packaging

All Linux distributions group software into packages, which can be defined as a collection of files and subdirectories comprising a product.

If all the software on the system has been installed using the package management utilities, installation, removal, checking integrity and upgrading of software becomes easier and more stable. Of course, there will be other files on the system, such as configuration files and user data, which generally will either be outside the packaging system, or be modified by system administrators from their original content.

There are two main packaging systems in use in Linux systems; **RPM** (used for example in all variations and descendants of Red Hat Enterprise Linux, Fedora and SUSE); and **deb** (used for example in Debian and Ubuntu).

It is important to note that there are always at least two levels of the package management software. The lower level simply installs, updates or removes a package, as in:

```
$ sudo rpm -ivh libaio-devel-0.3.109-12.el7.x86_64.rpm
```

which would install the development package associated with the asynchronous I/O library on a Red Hat-derived system. We will shortly show the use of **dpkg** and **apt-get** to do these operations on Debian-derived distributions. However, this will fail if the actual library itself is not installed. Thus, one would have to install them in the proper order, or both at the same time, as in:

```
$ sudo rpm -ivh libaio-devel-0.3.109-12.el7.x86_64.rpm libaio-l-0.3.109-12.el7
  .x86_64.rpm
```

This can be a process which is both error-prone and frustrating, as you can easily get into dependency hell, where each time you add a package, you find another one missing, and the **rpm** command is not robust enough to always tell you what package it is you need. Furthermore, you have to download the packages first, and supply the full name, including version and architecture, both of which are painful steps.

The higher level utilities handle this automatically. For example, on Red Hat-based systems, the command:

```
$ sudo yum install libaio-devel
```

will resolve any needed dependencies and then download all needed packages from one or more repositories the system has been configured to utilize, and then install them in the proper order. Likewise, the command:

```
$ sudo yum remove libaio
```

will not only remove **libaio** but also **libaio-devel**, since it cannot work without the base package. Obviously, you have to be careful when removing packages, as a cascading effect can happen, but **yum** will always give you a chance to change your mind before doing it.

Each distribution has such higher level management utilities. For example, **deb**-based systems have **apt-get** and **apt-cache**, while SUSE-based systems have **zypper**. They also have graphical utilities for package management which can insulate you from the command line. Recent Fedora systems use a newer program called **dnf** in place of **yum**.

There are a lot of holy wars in the Linux community about which packaging system is the best and there are others besides these two, both older ones and newer cutting edge ones. Often, the criticisms of the various methods are mistakenly applied to the lower level commands, when they should be applied to the higher level ones that deal with dependencies, etc.

Of course, all distributions have graphical system administration utilities for package management, that can do all of this without resort to the low-level commands, but once you know the basic commands, working at the command line tends to be significantly quicker.

Upgrading and Patching

From time to time, software on the system must be upgraded (or updated) or patched for one or more of the following reasons:

- Incorporation of new features
- Optimization and performance improvements
- Security and bug fixes.

In many operating systems, such patching can be a laborious and error-prone task, which may require frequent reboots. These can be extremely disruptive on server systems, and system administrators have learned to be very cautious when deploying patches.

Linux does not employ a patching model. Rather, it installs complete new packages. For example, on RPM-based systems, the command:

```
$ sudo rpm -Uvh libaio-devel-0.3.109-12.el7.x86_64.rpm
```

will put in the newest version; a similar command, using **dpkg**, can be used on Debian-based systems. Even better, depending on the distribution, one of the following commands:

```
$ sudo yum update libaio-devel
$ sudo zypper update libaio-devel
$ sudo apt-get upgrade libaio-dev
```

will go out and check if an update is available, download it and install it, and, at the same time, update any other installed packages that require synchronization. The commands:

```
$ sudo yum update
$ sudo zypper update
$ sudo apt-get update
$ sudo apt-get dist-upgrade
```

will update (and/or upgrade) all packages on the system. Note that Debian-based systems require **update** to first synchronize repository information before updating the entire package system.

In order to avoid large downloads, most recent distributions can use features such as delta rpms, which can download a smaller binary patch file, and then use that to recreate the full rpm. However, when bandwidth is high, the time spent on reconstruction of the full rpm may constitute a net minus.

All distributions have configurable background daemons that run at specified intervals to check for updates, and then either install them automatically, or ask for approval.

The only time you should ever need to reboot a Linux system during the update process is when the kernel itself is updated. This is a feature whose beauty cannot be overemphasized.

The following table lists the basic packaging operations and their **rpm** and **deb**-based equivalents. The **zypper**-based commands are almost identical to the **yum** ones; look at the man page for **zypper** on SUSE-based systems.

The following table lists the basic packaging operations and their **rpm** and **deb**-based equivalents. The **zypper**-based commands are almost identical to the **yum** ones; look at the man page for **zypper** on SUSE-based systems.

Operation	RPM	deb
Install a package	**rpm -i foo.rpm**	**dpkg --install foo.deb**
Install a package with dependencies from repository	**yum install foo**	**apt-get install foo**
Remove a package	**rpm -e foo.rpm**	**dpkg --remove foo.deb**
Remove a package and dependencies using a repository	**yum remove foo**	**apt-get remove foo**
Update package to a newer version	**rpm -U foo.rpm**	**dpkg --install foo.deb**
Update package using repository and resolving dependencies	**yum update foo**	**apt-get install foo**
Update entire system	**yum update**	**apt-get dist-upgrade**
Show all installed packages	**rpm -qa or yum list installed**	**dpkg --list**
Get information about an installed package including files	**rpm -qil foo**	**dpkg --listfiles foo**
Show available packages with "foo" in name	**yum list foo**	**apt-cache search foo**
Show all available packages	**yum list**	**apt-cache dumpavail foo**
What package does a file belong to?	**rpm -qf file**	**dpkg --search file**

Recent Fedora systems have replaced **yum** with a new program called **dnf**. The basic commands are the same as those with **yum**. However, advanced commands can be different or missing. The intention is to eventually replace **yum**; for now, when you issue **yum** commands, a warning is printed out and the equivalent **dnf** command is displayed and carried out.

User Directories, Environments, etc

On Linux systems, user directories are conventionally placed under **/home**, as in **/home/coop**, **/home/student**, etc. All personal configuration, data, and executable programs are placed in this directory hierarchy.

On other UNIX-like operating systems, the concept of the **/home** directory tree exists, but can be subtly different. For example, on Solaris, user directories are created in **/export/home** and then, the **automount** facility will eventually mount them in **/home**.

This is because the usual situation is that the home directory may be anywhere on a corporate network, probably on an NFS server, and the home directory will be mounted automatically upon first use.

Linux has these same automount facilities, but many users are not even aware of them, and on self-contained systems, the concept of NFS mounts will probably not apply.

A given user can always substitute the environment variable **HOME** for their root directory, or the shorthand **~**; i.e. the following are equivalent:

```
1  $ ls -l $HOME/public_html
2  $ ls -l ~/public_html
```

There is one very important exception: the home directory for the root user on Linux systems is always placed in **/root**. Other system-provided accounts (such as **daemon** and **bin**) can also have directories in locations other than **/home**.

Logging File

System log files are essential for monitoring and troubleshooting. In Linux, these messages appear in various files under **/var/log**.

Ultimate control of how messages are dealt with is controlled by the **syslogd** daemon (usually **rsyslogd** on modern systems) common to many UNIX-like operating systems. The newer systemd-based systems can use **journalctl** instead, but usually retain **syslogd** and cooperate with it.

Important messages are sent not only to the logging files, but also to the system console window; if you are not running X or are at a virtual terminal, you will see them directly there as well. In addition, these messages will be copied to **/var/log/messages** (or to **/var/log/syslog** on Ubuntu), but if you are running X, you have to take some steps to view them.

A good way to see them is to open a terminal window, and in that window, type **tail -f /var/log/messages**. On a GNOME desktop, you can also access the messages by clicking on *System* > *Administration* > *System Log or Applications* > *System Tools* > *Log File Viewer* in your Desktop menus, and other desktops have similar links you can locate.

In order to keep log files from growing without bound, the **logrotate** program is run periodically and keeps four previous copies (by default) of the log files (optionally compressed) and is controlled by **/etc/logrotate.conf**.

Here are some of the important log files found under **/var/log**:

File	Purpose
boot.log	System boot messages
dmesg	Kernel messages saved after boot. To see the current contents of the kernel message buffer, type **dmesg**
messages or **syslog**	All important system messages
secure	Security related messages

Basics of Users and Groups

Linux Users

- All Linux users are assigned a unique user ID, which is just an integer, as well as one or more group IDs (one of which is the default one and is the same as the user ID)
- The normal prescription is that normal users start with a user ID of 1000 and then go up from there
- These numbers are associated with more convenient strings, or names, through the files **`/etc/passwd`** and **`/etc/group`**
    - For example, the first file may contain:
      `george:x:1000:1000:George Metesky:/home/george:/bin/bash`
    - And the second one:
      `george:x:1000:`

/etc/passwd

- **/etc/passwd** (**george:x:1000:1000:George Metesky:/home/george:/bin/bash**) contains some important pieces of information, separated by colons:
  - Account name: (**george**)
    The name of the user on the system, which should not contain capital letters
  - Password: (**x**)
    This can be an encrypted password, an asterisk, or an x depending on how security is set up on the system
  - User ID: (**1000**)
    The numerical value for the User ID
  - Group ID: (**1000**)
    The numerical value for the primary Group ID

- **/etc/passwd** (**george:x:1000:1000:George Metesky:/home/george:/bin/bash**) contains some important pieces of information, separated by colons:
  - Full user name: (**George Metesky**)
    This field can be used for other purposes, but is almost always the full user name
  - Directory: (**/home/george**)
    The user's home directory
  - Shell: (**/bin/bash**)
    The user's default shell, which is the program run when logging in; if you see a **/sbin/nologin** or any non-executable program, this means the user cannot directly login (this is used for a lot of system daemons)
- If you look at **/etc/passwd**, you will see that almost all entries do not correspond to real users in the normal sense, but are special entities used for certain system utilities and functions

/etc/group

- **/etc/group** (**fuse:x:106:root,george**) is also straightforward and says that the **fuse** group, with numerical Group ID **106**, has as members **root** and **george**
- Groups are used to establish a set of users who have common interests for the purposes of access rights, privileges, and security considerations
- Access rights to files (and devices) are granted on the basis of the user and the group it belongs to

Adding and Removing Users and Groups

Adding a new user is done with **useradd** and removing an existing user is done with **userdel**. In the simplest form, an account for the new user **bjmoose** would be done with:

```
$ sudo /usr/sbin/useradd bjmoose
```

which, by default, sets the home directory to **/home/bjmoose**, populates it with some basic files (copied from **/etc/skel**), adds a line to **/etc/passwd** such as:

```
bjmoose:x:1002:1002::/home/bjmoose:/bin/bash
```

and sets the default shell to **/bin/bash**.

Additional options can be specified to change these properties, and to set others, such as the user name, etc. (see man **useradd**).

Before the account can be used, a password must be set. This can be done with the **-p** option to **useradd**, or by doing:

```
$ sudo passwd bjmoose
```

which will then prompt for adding a password.

Note that only the superuser, or root, has the right to establish (or remove) an account.

Removing a user account is as easy as:

```
$ sudo /usr/sbin/userdel bjmoose
```

However, this will leave the **/home/bjmoose** directory intact. This might be useful if it is a temporary inactivation, for example. To remove the home directory while removing the account, you need to use the **-r** option to **userdel**.

You can change the user's characteristics after the account has been established with **usermod**. For example, you could use the **-d** option to change the home directory, or the **-p** option to change the password.

Adding a new group is done with **groupadd**:

```
$ sudo /usr/sbin/groupadd anewgroup
```

establishes the group **anewgroup** with default properties. The group can be removed with:

```
$ sudo /usr/sbin/groupdel anewgroup
```

Adding a user to an already existing group is done with **usermod**. For example, you would first look at what groups the user already belongs to:

```
$ groups bjmoose
bjmoose : bjmoose
```

and then, add the new group:

```
$ sudo /usr/sbin/usermod -aG anewgroup bjmoose
$ groups bjmoose
bjmoose: rjsquirrel anewgroup
```

Once again, these utilities must be run as superuser or root, and update **/etc/group** as necessary. The **groupmod** utility can be used to change the group's properties, most often the numerical Group ID with the **-g** option, or its name with the **-m** option.

Removing a user from the group is somewhat trickier. The **-G** option to **usermod** must be given a complete list of groups. Thus, if you do:

```
$ sudo /usr/sbin/usermod -G rjsquirrel rjsquirrel
$ groups rjsquirrel
rjsquirrel : rjsquirrel
```

only the **rjsquirrel** group will be left.

An additional command, **id**, can be used to quickly glimpse user information. With no argument, it gives information about the current user, as in:

```
$ id
uid=1000(george) gid=1000(george) groups=106(fuse),1000(george)
```

If given the name of another user as an argument, **id** will report information about that other user.

Files, Users and Permissions

Suppose you have a file and obtain a detailed listing of its properties:

```
$ ls -lF file
-rwxr-x--x 1 coop coop 42 Jun 18 13:59 some_file*
```

After the initial dash, there are nine letters, in three groups of three, that indicate read, write and execute permissions for owner, group, and world. In the above example, the owner of the file can read, write and execute, all members of the group can read or execute, and all others (in the world) can only execute.

These file access permissions are a critical part of the Linux security system. Any request to access a file requires comparison of the credentials and identity of the requesting user to those of the owner of the file.

This authorization is granted depending on one of these three sets of permissions, in the following order:

- If the requester is the file owner, the file owner permissions are used.
- Otherwise, if the requester is in the group that owns the files, the group permissions are examined.
- If that does not succeed, the world permissions are examined.

Note that permissions can be changed with **chmod** and ownership with **chown**.

One user is special; the superuser or root user, who has access to all files on the system. This is essentially the equivalent to the administrator account, or privilege, in other operating systems.

Linux contains a full implementation of POSIX ACLs (Access Control Lists) which extends the simpler user, group, world and read, write, execute model.

Particular privileges can be granted to specific users or groups of users when accessing certain objects or classes of objects.

While the Linux kernel enables the use of ACLs, it still must be implemented as well in the particular filesystem. All major filesystems used in modern Linux distributions incorporate the ACL extensions.

root (Super) user, su and sudo

It is possible to enter the system as the root user either for a series of operations or only for one. As a general rule, you should assume so-called root privileges only when absolutely necessary and for as short a time as necessary.

In order to temporarily sign on as another user, you can use the **su** command, as in the following examples:

```
$ su anotheruser
$ su root
$ su
```

You will be prompted for the password of the user whose name was specified. If you do not give a user name (as in the third example), root will be assumed.

The superuser session ends when you type **exit** in the shell.

If you use a naked dash as an option, as in:

```
$ su -
```

there is a subtle difference; you are signed into a login shell, which means your working directory will shift to the home directory of the account you are logging into, paths will change, etc.

If you use the **-c** option as in:

```
$ su root -c ls
$ su - root -c ls
```

you execute only one command, in this case **ls**. In the first case, this will be in the current working directory, in the second, in the root's home directory.

Suppose a normal user needs temporary root privilege to execute a command, say to put a file in a directory that requires root privilege. You can do that with the **su** command, but there is one obvious drawback; the user needs to have the root password in order to do this.

Once you have made the root password known to a normal user, you have abandoned all notions of security. While this may be an acceptable day-to-day method on a system on which you are the only normal user and you are trying to respect good system hygiene by avoiding privilege escalation except when absolutely necessary, there is a better method involving the **sudo** command.

To use **sudo** you merely have to do:

```
$ sudo -u anotheruser command
$ sudo command
```

where in the second form, the implicit user is root. While this resembles doing **su -c**, it is quite different in that the user's own password is required; **su** requires the other user's password (often that of root).

However, this will not work unless the superuser has already updated **/etc/sudoers** to grant you permission to use the **sudo** command. Furthermore, it is possible to limit exactly which subset of commands a particular user or group has access to, and to permit usage with a password prompt.

It is not recommended to edit this file in a normal manner. You should use either the **visudo** or **sudoedit** programs (as root) instead, as in:

```
# /usr/sbin/visudo
# /usr/sbin/sudoedit /etc/sudoers
```

because they check the resulting edited file to make sure it has no errors in it before exiting. Otherwise, you can wind up in a difficult-to-fix situation, especially on Linux distributions such as Ubuntu, which hide the root account and rely heavily on the use of **sudo**.

Rather than discussing the details of how to do such fine tuning, we recommend you read this file as it is self-documenting, or do **man sudoers**.

The simplest line you could add to this file would be (for user **student**):

```
student ALL=(ALL) ALL
```

which would let the user have all normal root privileges.

On all recent Linux distributions, you should not modify the file **/etc/sudoers**. Instead, there is a sub-directory **/etc/sudoers.d** in which you can create individual files for individual users.

Thus, you can simply make a short file in **/etc/sudoers.d** containing the above line, and then give it proper permissions as in:

```
$ chmod 440 /etc/sudoers.d/student
```

Note that some Linux distributions may require **chmod 400 /etc/sudoers.d/student**.

If a file named **/tmp/rootfile** is owned by root, the command:

```
$ sudo echo hello > /tmp/rootfile
```

will fail due to permission problems.

The proper way to do this would be:

```
$ sudo bash -c "echo hello > /tmp/rootfile"
```

Do you see why?

Some Linux distributions, notably Ubuntu, do not work with root user accounts in the traditional UNIX fashion.

Instead, there appears to be only a normal user account, and the same password is used to log into the system as a normal user, and to use **sudo**. In fact, there is no direct **su** command. However, the equivalent is easily accomplished through the command: **sudo su**.

To some UNIX traditionalists, this method of dealing with the root account is a bad idea, and some attribute its use to an attempt to make Windows users more comfortable when they move to Linux.