# 2009/10
# IT Training Programme for SMEs in General Industries

## << T09 – Unix/Linux 入門>>

**資助機構:**

香港特別行政區政府
政府資訊科技總監辦公室
**Office of the Government**
**Chief Information Officer**
**The Government of the HKSAR**

**主辦機構:**

MEIIA
製造業創滙商會
Manufacturing Enterprise Integration and Innovation Association

**協辦機構:**

Hong Kong Productivity Council
香港生產力促進局



## 1. Introduction to Unix

### Unix

The Unix operating system was conceived and implemented in 1969 at AT&T's Bell Laboratories in the United States by Ken Thompson, Dennis Ritchie, Douglas McIlroy, and Joe Ossanna. It was first released in 1971 and was initially entirely written in assembly language, a common practice at the time. Later, in a key pioneering approach in 1973, Unix was re-written in the programming language C by Dennis Ritchie, (with exceptions to the kernel and I/O). The availability of an operating system written in a high-level language allowed easier portability to different computer platforms and Unix became widely adopted by academic institutions and businesses.

### GNU

The GNU Project, started in 1983 by Richard Stallman, had the goal of creating a "*complete Unix-compatible software system*" composed entirely of free software. Work began in 1984.[15] Later, in 1985, Stallman created the Free Software Foundation and wrote the GNU General Public License (GNU GPL) in 1989. By the early 1990s, many of the programs required in an operating system (such as libraries, compilers, text editors, a Unix shell, and a windowing system) were completed, although low-level elements such as device drivers, daemons, and the kernel were stalled and incomplete.[16] Linus Torvalds has said that if the GNU kernel had been available at the time (1991), he would not have decided to write his own.

## MINIX

Andrew S. Tanenbaum, author of the MINIX operating system

MINIX was an inexpensive minimal Unix-like operating system, designed for education in computer science, written by Andrew S. Tanenbaum (now MINIX is free and redesigned also for “serious” use).

In 1991 while attending the University of Helsinki, Torvalds began to work on a non-commercial replacement for MINIX,[18] which would eventually become the Linux kernel.

Torvalds began the development of Linux on MINIX and applications written for MINIX were also used under Linux. Later Linux matured and it became possible for Linux to be developed under itself.Also GNU applications replaced all MINIX ones because, with code from the GNU system freely available, it was advantageous if this could be used with the fledgling OS. Code licensed under the GNU GPL can be used in other projects, so long as they also are released under the same or a compatible license. In order to make the Linux kernel compatible with the components from the GNU Project, Torvalds initiated a switch from his original license (which prohibited commercial redistribution) to the GNU GPL.Developers worked to integrate GNU components with Linux to make a fully functional and free operating system.



## LINUX

**Linux** is a generic term referring to Unix-like computer operating systems based on the Linux kernel. Their development is one of the most prominent examples of free and open source software collaboration;

typically all the underlying source code can be used, freely modified, and redistributed, both commercially and non-commercially, by anyone under licenses such as the GNU GPL.

Linux is predominantly known for its use in servers, although can be installed on a wide variety of computer hardware, ranging from embedded devices, mobile phones and even some watches to mainframes and supercomputers. Linux distributions, installed on both desktop and laptop computers, have become increasingly commonplace in recent years, partly owing to the popular Ubuntu distribution and the emergence of netbooks.

The name "Linux" comes from the Linux kernel, originally written in 1991 by Linus Torvalds. The full operating system usually comprises components such as utilities and libraries from the GNU Project (announced in 1983 by Richard Stallman), the X Window System, the GNOME and KDE desktop environments, and the Apache HTTP Server. Commonly-used applications with desktop Linux systems include the Mozilla Firefox web-browser and the OpenOffice.org office application suite. The GNU contribution is the basis for the Free Software Foundation's preferred name *GNU/Linux*.

# What is Linux Shell

Computer understand the language of 0's and 1's called binary language.

In early days of computing, instruction are provided using binary language, which is difficult for all of us, to read and write. So in Os there is special program called Shell. Shell accepts your instruction or commands in English (mostly) and if its a valid command, it is pass to kernel.

Shell is a user program or it's environment provided for user interaction. Shell is an command language interpreter that executes commands read from the standard input device (keyboard) or from a file.

Shell is not part of system kernel, but uses the system kernel to execute programs, create files etc.

Several shell available with Linux including:

| Shell Name | Developed by | Where | Remark |
|---|---|---|---|
| BASH ( Bourne-Again SHell ) | Brian Fox and Chet Ramey | Free Software Foundation | Most common shell in Linux. It's Freeware shell. |
| CSH (C SHell) | Bill Joy | University of California (For BSD) | The C shell's syntax and usage are very similar to the C programming language. |
| KSH (Korn SHell) | David Korn | AT & T Bell Labs | -- |
| TCSH | See the man page. Type $ man tcsh | -- | TCSH is an enhanced but completely compatible version of the Berkeley UNIX C shell (CSH). |

***Tip:*** To find all available shells in your system type following command:
**$ cat /etc/shells**

***Note*** that each shell does the same job, but each understand a different command syntax and provides different built-in functions.



In MS-DOS, Shell name is COMMAND.COM which is also used for same purpose, but it's not as powerful as our Linux Shells are!

Any of the above shell reads command from user (via Keyboard or Mouse) and tells Linux Os what users want. If we are giving commands from keyboard it is called command line interface ( Usually in-front of $ prompt, This prompt is depend upon your shell and Environment that you set or by your System Administrator, therefore you may get different prompt ).

***Tip:*** To find your current shell type following command
**$ echo $SHELL**

# How to use Shell

To use shell (You start to use your shell as soon as you log into your system) you have to simply type commands.

# What is Shell Script ?

Normally shells are interactive. It means shell accept command from you (via keyboard) and execute them. But if you use command one by one (sequence of 'n' number of commands) , the you can store this sequence of command to text file and tell the shell to execute this text file instead of entering the commands. This is know as ***shell script***.

Shell script defined as:
"*Shell Script is **series of command** written **in plain text file**. Shell script is just like batch file is MS-DOS but have more power than the MS-DOS batch file.*"

## 2. Text Editing

### Vi(m)

Vim stands for "Vi IMproved". It used to be "Vi IMitation", but there are so many improvements that a name change was appropriate. Vim is a text editor which includes almost all the commands from the UNIX program **vi** and a lot of new ones.

Commands in the **vi** editor are entered using only the keyboard, which has the advantage that you can keep your fingers on the keyboard and your eyes on the screen, rather than moving your arm repeatedly to the mouse. For those who want it, mouse support and a GUI version with scrollbars and menus can be activated.

We will refer to **vi** or **vim** throughout this book for editing files, while you are of course free to use the editor of your choice. However, we recommend to at least get the **vi** basics in the fingers, because it is the standard text editor on almost all UNIX systems, while **emacs** can be an optional package. There may be small differences between different computers and terminals, but the main point is that if you can work with **vi**, you can survive on any UNIX system.

### Moving through the text

Moving through the text is usually possible with the arrow keys. If not, try:

- **h** to move the cursor to the left
- **l** to move it to the right
- **k** to move up
- **j** to move down

SHIFT-G will put the prompt at the end of the document.



# Basic operations

These are some popular **vi** commands:

- **n dd** will delete n lines starting from the current cursor position.
- **n dw** will delete n words at the right side of the cursor.
- **x** will delete the character on which the cursor is positioned
- **:n** moves to line n of the file.
- **:w** will save (write) the file
- **:q** will exit the editor.
- **:q!** forces the exit when you want to quit a file containing unsaved changes.
- **:wq** will save and exit
- **:w newfile** will save the text to newfile.
- **:wq!** overrides read-only permission (if you have the permission to override permissions, for instance when you are using the *root* account.
- **/astring** will search the string in the file and position the cursor on the first match below its position.
- **/** will perform the same search again, moving the cursor to the next match.
- **:1, $s/word/anotherword/g** will replace word with anotherword throughout the file.
- **yy** will copy a block of text.
- **n p** will paste it n times.
- **:recover** will recover a file after an unexpected interruption.

### Commands that switch the editor to insert mode

- **a** will append: it moves the cursor one position to the right before switching to insert mode
- **i** will insert
- **o** will insert a blank line under the current cursor position and move the cursor to that line.

Pressing the **Esc** key switches back to command mode. If you're not sure what mode you're in because you use a really old version of **vi** that doesn't display an "INSERT" message, type **Esc** and you'll be sure to return to command mode. It is possible that the system gives a little alert when you are already in command mode when hitting **Esc**, by beeping or giving a visual bell (a flash on the screen). This is normal behavior.

# 3. The File System

**EXT2**

The **ext2** or **second extended filesystem** is a file system for the Linux kernel. It was initially designed by Rémy Card as a replacement for the extended file system (ext).

The canonical implementation of ext2 is the ext2fs filesystem driver in the Linux kernel. Other implementations (of varying quality and completeness) exist in GNU Hurd, Mac OS X (third-party), Darwin (same third-party as Mac OS X but untested), some BSD kernels, in Atari MiNT, and as third-party Microsoft Windows drivers.

ext2 was the default filesystem in several Linux distributions, including Debian and Red Hat Linux, until supplanted more recently by ext3, which is almost completely compatible with ext2 and is a journaling file system. ext2 is still the filesystem of choice for flash-based storage media (such as SD cards, SSDs, and USB flash drives) since its lack of a journal minimizes the number of writes and flash devices have only a limited number of write cycles.

## ext2 data structures

The space in ext2 is split up in blocks, and organized into block groups, analogous to cylinder groups in the Unix File System. This is done to reduce external fragmentation and minimize the number of disk seeks when reading a large amount of consecutive data.

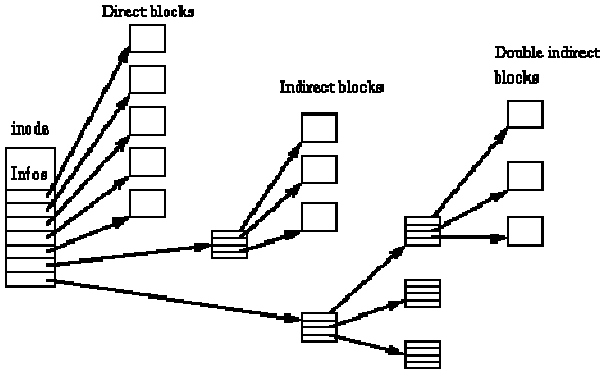
Each block group may contain a copy of the superblock and block group descriptor table, and all block groups contain a block bitmap, an inode bitmap, an inode table and followed by the actual data blocks.

The superblock contains important information that is crucial to the booting of the operating system, thus backup copies are made in multiple block groups in the file system. However, typically only the first copy of it, which is found at the first block of the file system, is used in the booting.

The group descriptor stores the location of the block bitmap, inode bitmap and the start of the inode table for every block group and these, in turn are stored in a group descriptor table.



Example of ext2 inode structure:



**EXT3**

The **ext3** or **third extended filesystem** is a journaled file system that is commonly used by the Linux kernel. It is the default file system for many popular Linux distributions. Stephen Tweedie first revealed that he was working on extending ext2 in *Journaling the Linux ext2fs Filesystem* in a 1998 paper and later in a February 1999 kernel mailing list posting, and the filesystem was merged with the mainline Linux kernel in November 2001 from 2.4.15 onward.[2][3][4] Its main advantage over ext2 is journaling which improves reliability and eliminates the need to check the file system after an unclean shutdown. Its successor is ext4.

## Advantages

Although its performance (speed) is less attractive than competing Linux filesystems such as JFS, ReiserFS and XFS, it has a significant advantage in that it allows in-place upgrades from the ext2 file system without having to back up and restore data. Ext3 also uses less CPU power than ReiserFS and XFS.[5] It is also considered safer than the other Linux file systems due to its relative simplicity and wider testing base.[*citation needed*]

The ext3 file system adds, over its predecessor:

- A Journaling file system
- Online file system growth
- Htree indexing for larger directories. An HTree is a specialized version of a B-tree (not to be confused with the H tree fractal)[6].

Without these, any ext3 file system is also a valid ext2 file system. This has allowed well-tested and mature file system maintenance utilities for maintaining and repairing ext2 file systems to also be used with ext3 without major changes. The ext2 and ext3 file systems share the same standard set of utilities, e2fsprogs, which includes a fsck tool. The close relationship also makes conversion between the two file systems (both forward to ext3 and backward to ext2) straightforward.

While in some contexts the lack of "modern" filesystem features such as dynamic inode allocation and extents could be considered a disadvantage, in terms of recoverability this gives ext3 a significant advantage over file systems with those features. The file system metadata is all in fixed, well-known locations, and there is some redundancy inherent in the data structures that may allow ext2 and ext3 to be recoverable in the face of significant data corruption, where tree-based file systems may not be recoverable.



## Sorts of files

Most files are just files, called *regular* files; they contain normal data, for example text files, executable files or programs, input for or output from a program and so on.

While it is reasonably safe to suppose that everything you encounter on a Linux system is a file, there are some exceptions.

- *Directories*: files that are lists of other files.
- *Special files*: the mechanism used for input and output. Most special files are in `/dev`, we will discuss them later.
- *Links*: a system to make a file or directory visible in multiple parts of the system's file tree. We will talk about links in detail.
- *(Domain) sockets*: a special file type, similar to TCP/IP sockets, providing inter-process networking protected by the file system's access control.
- *Named pipes*: act more or less like sockets and form a way for processes to communicate with each other, without using network socket semantics.

The `-l` option to **ls** displays the file type, using the first character of each input line:

```
jaime:~/Documents> ls -l
total 80
-rw-rw-r--   1 jaime   jaime   31744 Feb 21 17:56 intro Linux.doc
-rw-rw-r--   1 jaime   jaime   41472 Feb 21 17:56 Linux.doc
drwxrwxr-x   2 jaime   jaime    4096 Feb 25 11:50 course
```

Table File types in a long list

| Symbol | Meaning |
|---|---|
| - | Regular file |
| D | Directory |
| l | Link |
| c | Special file |
| s | Socket |
| p | Named pipe |
| b | Block device |



## Partition layout and types

There are two kinds of major partitions on a Linux system:

- *data partition*: normal Linux system data, including the *root partition* containing all the data to start up and run the system; and
- *swap partition*: expansion of the computer's physical memory, extra memory on hard disk.

Most systems contain a root partition, one or more data partitions and one or more swap partitions. Systems in mixed environments may contain partitions for other system data, such as a partition with a FAT or VFAT file system for MS Windows data.

Most Linux systems use **fdisk** at installation time to set the partition type. As you may have noticed during the exercise from Chapter 1, this usually happens automatically. On some occasions, however, you may not be so lucky. In such cases, you will need to select the partition type manually and even manually do the actual partitioning. The standard Linux partitions have number 82 for swap and 83 for data, which can be journaled (ext3) or normal (ext2, on older systems). The **fdisk** utility has built-in help, should you forget these values.

Apart from these two, Linux supports a variety of other file system types, such as the relatively new Reiser file system, JFS, NFS, FATxx and many other file systems natively available on other (proprietary) operating systems.

The standard root partition (indicated with a single forward slash, */*) is about 100-500 MB, and contains the system configuration files, most basic commands and server programs, system libraries, some temporary space and the home directory of the administrative user. A standard installation requires about 250 MB for the root partition.

The rest of the hard disk(s) is generally divided in data partitions, although it may be that all of the non-system critical data resides on one partition, for example when you perform a standard workstation installation. When non-critical data is separated on different partitions, it usually happens following a set pattern:

- a partition for user programs (*/usr*)
- a partition containing the users' personal data (*/home*)
- a partition to store temporary data like print- and mail-queues (*/var*)
- a partition for third party and extra software (*/opt*)

Once the partitions are made, you can only add more. Changing sizes or properties of existing partitions is possible but not advisable.

## Mount points

All partitions are attached to the system via a mount point. The mount point defines the place of a particular data set in the file system. Usually, all partitions are connected through the *root* partition. On this partition, which is indicated with the slash (/), directories are created. These empty directories will be the starting point of the partitions that are attached to them. An example: given a partition that holds the following directories:

```
videos/         cd-images/      pictures/
```

We want to attach this partition in the filesystem in a directory called `/opt/media`. In order to do this, the system administrator has to make sure that the directory `/opt/media` exists on the system. Preferably, it should be an empty directory. How this is done is explained later in this chapter. Then, using the **mount** command, the administrator can attach the partition to the system. When you look at the content of the formerly empty directory `/opt/media`, it will contain the files and directories that are on the mounted medium (hard disk or partition of a hard disk, CD, DVD, flash card, USB or other storage device).

During system startup, all the partitions are thus mounted, as described in the file `/etc/fstab`. Some partitions are not mounted by default, for instance if they are not constantly connected to the system, such like the storage used by your digital camera. If well configured, the device will be mounted as soon as the system notices that it is connected, or it can be user-mountable, i.e. you don't need to be system administrator to attach and detach the device to and from the system.



## The chmod command

### Table File protection with chmod

| Command | Meaning |
|---|---|
| **chmod *400* `file`** | To protect a file against accidental overwriting. |
| **chmod *500* `directory`** | To protect yourself from accidentally removing, renaming or moving files from this directory. |
| **chmod *600* `file`** | A private file only changeable by the user who entered this command. |
| **chmod *644* `file`** | A publicly readable file that can only be changed by the issuing user. |
| **chmod *660* `file`** | Users belonging to your group can change this file, others don't have any access to it at all. |
| **chmod *700* `file`** | Protects a file against any access from other users, while the issuing user still has full access. |
| **chmod *755* `directory`** | For files that should be readable and executable by others, but only changeable by the issuing user. |
| **chmod *775* `file`** | Standard file sharing mode for a group. |
| **chmod *777* `file`** | Everybody can do everything to this file. |

### File permissions

| Who\What | r(ead) | w(rite) | (e)x(ecute) |
|---|---|---|---|
| u(ser) | 4 | 2 | 1 |
| g(roup) | 4 | 2 | 1 |
| o(ther) | 4 | 2 | 1 |

## 4. Text File Manipulation

In computing, **regular expressions**, also referred to as **regex** or **regexp**, provide a concise and flexible means for matching strings of text, such as particular characters, words, or patterns of characters. A regular expression is written in a formal language that can be interpreted by a regular expression processor, a program that either serves as a parser generator or examines text and identifies parts that match the provided specification.

The following examples illustrate a few specifications that could be expressed in a regular expression:

- The sequence of characters "car" in any context, such as "car", "cartoon", or "bicarbonate"
- The word "car" when it appears as an isolated word
- The word "car" when preceded by the word "blue" or "red"
- A dollar sign immediately followed by one or more digits, and then optionally a period and exactly two more digits

Regular expressions can be much more complex than these examples.

Regular expressions are used by many text editors, utilities, and programming languages to search and manipulate text based on patterns. For example, Perl, Ruby and Tcl have a powerful regular expression engine built directly into their syntax. Several utilities provided by Unix distributions—including the editor ed and the filter grep—were the first to popularize the concept of regular expressions.

As an example of the syntax, the regular expression `\bex` can be used to search for all instances of the string "*ex*" that occur after "word boundaries" (signified by the `\b`). In layman's terms, `\bex` will find the matching string "ex" in two possible locations, (1) at the beginning of words, and (2) between two characters in a string, where one is a word character and the other is not a word character. Thus, in the string "Texts for experts," `\bex` matches the "*ex*" in "experts" but not in "Texts" (because the "*ex*" occurs inside a word and not immediately after a word boundary).

Many modern computing systems provide wildcard characters in matching filenames from a file system. This is a core capability of many command-line shells and is also known as globbing. Wildcards differ from regular expressions in generally only expressing very limited forms of alternatives.



## Basic concepts

Boolean "or"

A vertical bar separates alternatives. For example, `gray|grey` can match "*gray*" or "*grey*".

Grouping

Parentheses are used to define the scope and precedence of the operators (among other uses). For example, `gray|grey` and `gr(a|e)y` are equivalent patterns which both describe the set of "*gray*" and "*grey*".

Quantification

A quantifier after a token (such as a character) or group specifies how often that preceding element is allowed to occur. The most common quantifiers are the question mark ?, the asterisk * (derived from the Kleene star), and the plus sign +.

**?** The question mark indicates there is *zero or one* of the preceding element. For example, `colou?r` matches both "*color*" and "*colour*".

***** The asterisk indicates there are *zero or more* of the preceding element. For example, `ab*c` matches "*ac*", "*abc*", "*abbc*", "*abbbc*", and so on.

**+** The plus sign indicates that there is *one or more* of the preceding element. For example, `ab+c` matches "*abc*", "*abbc*", "*abbbc*", and so on, but not "*ac*".

# grep command syntax

```
grep 'word' filename
grep 'string1 string2'  filename
cat otherfile | grep 'something'
command | grep 'something'
```

# Use grep to search file

Search /etc/passwd for boo user:

```
$ grep boo /etc/passwd
```

You can force grep to ignore word case i.e match boo, Boo, BOO and all other combination with -i option:

```
$ grep -i "boo" /etc/passwd
```

# Use grep recursively

You can search recursively i.e. read all files under each directory for a string "192.168.1.5"

```
$ grep -r "192.168.1.5" /etc/
```

# Use grep to search words only

When you search for boo, grep will match fooboo, boo123, etc. You can force grep to select only those lines containing matches that form whole words i.e. match only boo word:

```
$ grep -w "boo" /path/to/file
```

# Use grep to search 2 different words

use egrep as follows:

```
$ egrep -w 'word1|word2' /path/to/file
```



# Count line when words has been matched

grep can report the number of times that the pattern has been matched for each file using -c (count) option:

```
$ grep -c 'word' /path/to/file
```

Also note that you can use -n option, which causes grep to precede each line of output with the number of the line in the text file from which it was obtained:

```
$ grep -n 'word' /path/to/file
```

# Grep invert match

You can use -v option to print inverts the match; that is, it matches only those lines that do not contain the given word. For example print all line that do not contain the word bar:

```
$ grep -v bar /path/to/file
```

## Sed

**sed** (*stream editor*) is a Unix utility that (a) parses text files and (b) implements a programming language which can apply textual transformations to such files. It reads input files line by line (sequentially), applying the operation which has been specified via the command line (or a *sed script*), and then outputs the line. It was developed from 1973 to 1974 as a Unix utility by Lee E. McMahon of Bell Labs,[1] and is available today for most operating systems.

# Samples

To delete a word from the file use:

```
sed '/yourword/d' yourfile > newfile
```

To delete two words for a file simultaneously use:

```
sed -e s/firstword//g -e s/secondword//g myfile > newfile
```

In the next example, sed, which usually only works on one line, removes newlines from sentences where the second sentence starts with one space. Consider the following text:

```
This is my cat
 my cat's name is betty
This is my dog
 my dog's name is frank
```

The sed script below will turn it into:

```
This is my cat my cat's name is betty
This is my dog my dog's name is frank
```



Here's the script:

```
sed 'N;s/\n / /;P;D;'
```

- (N) add the next line to the work buffer
- (s) substitute
- (/\n /) match: \n (newline character in Unix) and one space
- (/ /) replace with: one space
- (P) print the top line of the work buffer
- (D) delete the top line from the work buffer and run the script again

## Awk

The first awk

Let's go ahead and start playing around with awk to see how it works. At the command line, enter the following command:

```
$ awk '{ print }' /etc/passwd
```

You should see the contents of your /etc/passwd file appear before your eyes. Now, for an explanation of what awk did. When we called awk, we specified /etc/passwd as our input file. When we executed awk, it evaluated the print command for each line in /etc/passwd, in order. All output is sent to stdout, and we get a result identical to catting /etc/passwd. Now, for an explanation of the `{ print }` code block. In awk, curly braces are used to group blocks of code together, similar to C. Inside our block of code, we have a single print command. In awk, when a print command appears by itself, the full contents of the current line are printed.

Here is another awk example that does exactly the same thing:

```
$ awk '{ print $0 }' /etc/passwd
```

In awk, the $0 variable represents the entire current line, so `print` and `print $0` do exactly the same thing. If you'd like, you can create an awk program that will output data totally unrelated to the input data. Here's an example:

```
$ awk '{ print "" }' /etc/passwd
```

Whenever you pass the "" string to the print command, it prints a blank line. If you test this script, you'll find that awk outputs one blank line for every line in your /etc/passwd file. Again, this is because awk executes your script for every line in the input file. Here's another example:

```
$ awk '{ print "hiya" }' /etc/passwd
```

Running this script will fill your screen with hiya's. :)



Multiple fields

Awk is really good at handling text that has been broken into multiple logical fields, and allows you to effortlessly reference each individual field from inside your awk script. The following script will print out a list of all user accounts on your system:

```
$ awk -F":" '{ print $1 }' /etc/passwd
```

Above, when we called awk, we use the -F option to specify ":" as the field separator. When awk processes the `print $1` command, it will print out the first field that appears on each line in the input file. Here's another example:

```
$ awk -F":" '{ print $1 $3 }' /etc/passwd
```

Here's an excerpt of the output from this script:

```
halt7
operator11
root0
shutdown6
sync5
bin1
....etc.
```

As you can see, awk prints out the first and third fields of the /etc/passwd file, which happen to be the username and uid fields respectively. Now, while the script did work, it's not perfect -- there aren't any spaces between the two output fields! If you're used to programming in bash or python, you may have expected the `print $1 $3` command to insert a space between the two fields. However, when two strings appear next to each other in an awk program, awk concatenates them without adding an intermediate space. The following command will insert a space between both fields:

```
$ awk -F":" '{ print $1 " " $3 }' /etc/passwd
```

When you call print this way, it'll concatenate $1, " ", and $3, creating readable output. Of course, we can also insert some text labels if needed:

```
$ awk -F":" '{ print "username: " $1 "\t\tuid:" $3" }' /etc/passwd
```

This will cause the output to be:

```
username: halt     uid:7
username: operator uid:11
username: root     uid:0
username: shutdown uid:6
username: sync     uid:5
username: bin      uid:1
....etc.
```

---



# 5. Other File Processing Commands

Quickstart commands

| Command | Meaning |
|---|---|
| **Ls** | Displays a list of files in the current working directory, like the **dir** command in DOS |
| **cd** `directory` | change directories |
| **passwd** | change the password for the current user |
| **file** `filename` | display file type of file with name `filename` |
| **cat** `textfile` | throws content of `textfile` on the screen |
| **pwd** | display present working directory |
| **Exit** or **logout** | leave this session |
| **man** *command* | read man pages on **command** |
| **Info** *command* | read Info pages on **command** |
| **apropos** *string* | search the *whatis* database for strings |

Color-ls default color scheme

| Color | File type |
|---|---|
| blue | directories |
| red | compressed archives |
| white | text files |
| pink | images |
| cyan | links |
| yellow | devices |
| green | executables |
| flashing red | broken links |

### The head and tail commands

These two commands display the n first/last lines of a file respectively. To see the last ten commands entered:

```
tony:~> tail -10 .bash_history
locate configure | grep bin
man bash
cd
xawtv &
grep usable /usr/share/dict/words
grep advisable /usr/share/dict/words
info quota
man quota
echo $PATH
frm
```

**head** works similarly. The **tail** command has a handy feature to continuously show the last n lines of a file that changes all the time. This -f option is often used by system administrators to check on log files. More information is located in the system documentation files.



# Linking files

Since we know more about files and their representation in the file system, understanding links (or shortcuts) is a piece of cake. A link is nothing more than a way of matching two or more file names to the same set of file data. There are two ways to achieve this:

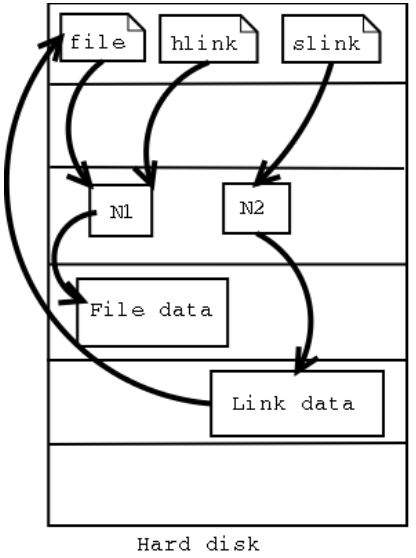
- Hard link: Associate two or more file names with the same inode. Hard links share the same data blocks on the hard disk, while they continue to behave as independent files.

  There is an immediate disadvantage: hard links can't span partitions, because inode numbers are only unique within a given partition.

- Soft link or symbolic link (or for short: symlink): a small file that is a pointer to another file. A symbolic link contains the path to the target file instead of a physical location on the hard disk. Since inodes are not used in this system, soft links can span across partitions.

The two link types behave similar, but are not the same, as illustrated in the scheme below:

**Figure 3-2. Hard and soft link mechanism**



Note that removing the target file for a symbolic link makes the link useless.



# The at command

The **at** command executes commands at a given time, using your default shell unless you tell the command otherwise (see the man page).

The options to **at** are rather user-friendly, which is demonstrated in the examples below:

```
steven@home:~> at tomorrow + 2 days
warning: commands will be executed using (in order) a) $SHELL
       b) login shell c) /bin/sh
at>  cat reports | mail myboss@mycompany
at> <EOT>
job 1 at 2001-06-16 12:36
```

Typing **Ctrl**+**D** quits the **at** utility and generates the "EOT" message.

Redirecting Input and Output.

```
steven@home:~> at 0237
warning: commands will be executed using (in order) a) $SHELL
       b) login shell c) /bin/sh
at>  cd new-programs
at>  ./configure; make
at> <EOT>
job 2 at 2001-06-14 02:00
```

## Cron and crontab

The cron system is managed by the **cron** daemon. It gets information about which programs and when they should run from the system's and users' crontab entries. Only the root user has access to the system crontabs, while each user should only have access to his own crontabs. On some systems (some) users may not have access to the cron facility.

At system startup the cron daemon searches `/var/spool/cron/` for crontab entries which are named after accounts in `/etc/passwd`, it searches `/etc/cron.d/` and it searches `/etc/crontab`, then uses this information every minute to check if there is something to be done. It executes commands as the user who owns the crontab file and mails any output of commands to the owner.

On systems using Vixie cron, jobs that occur hourly, daily, weekly and monthly are kept in separate directories in `/etc` to keep an overview, as opposed to the standard UNIX cron function, where all tasks are entered into one big file.

Example of a Vixie crontab file:

```
[root@blob /etc]# more crontab
SHELL=/bin/bash
PATH=/sbin:/bin:/usr/sbin:/usr/bin
MAILTO=root
HOME=/

# run-parts
# commands to execute every hour
01 * * * * root run-parts /etc/cron.hourly
# commands to execute every day
02 4 * * * root run-parts /etc/cron.daily
# commands to execute every week
22 4 * * 0 root run-parts /etc/cron.weekly
commands to execute every month
42 4 1 * * root run-parts /etc/cron.monthly
```



# 6. Networking Command

# Network configuration files

### etc/hosts

The `/etc/hosts` file always contains the *localhost* IP address, 127.0.0.1, which is used for interprocess communication. Never remove this line! Sometimes contains addresses of additional hosts, which can be contacted without using an external naming service such as DNS (the Domain Name Server).

A sample `hosts` file for a small home network:

```
# Do not remove the following line, or various programs
# that require network functionality will fail.
127.0.0.1       localhost.localdomain   localhost
192.168.52.10   tux.mylan.com           tux
192.168.52.11   winxp.mylan.com         winxp
```

Read more in **man hosts**.

### /etc/resolv.conf

The `/etc/resolv.conf` file configures access to a DNS serve

```
search mylan.com
nameserver 193.134.20.4
```

Read more in the `resolv.conf` man page.

### /etc/nsswitch.conf

The `/etc/nsswitch.conf` file defines the order in which to contact different name services. For Internet use, it is important that *dns* shows up in the "hosts" line:

```
[bob@tux ~] grep hosts /etc/nsswitch.conf
hosts:  files dns
```

This instructs your computer to look up hostnames and IP addresses first in the `/etc/hosts` file, and to contact the DNS server if a given host does not occur in the local `hosts` file. Other possible name services to contact are LDAP, NIS and NIS+.

More in **man nsswitch.conf**.



# The ip command

The distribution-specific scripts and graphical tools are front-ends to **ip** (or **ifconfig** and **route** on older systems) to display and configure the kernel's networking configuration.

The **ip** command is used for assigning IP addresses to interfaces, for setting up routes to the Internet and to other networks, for displaying TCP/IP configurations etcetera.

The following commands show IP address and routing information:

```
benny@home benny> ip addr show
1: lo: <LOOPBACK,UP> mtu 16436 qdisc noqueue
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 brd 127.255.255.255 scope host lo
    inet6 ::1/128 scope host
2: eth0: <BROADCAST,MULTICAST,UP> mtu 1500 qdisc pfifo_fast qlen 100
    link/ether 00:50:bf:7e:54:9a brd ff:ff:ff:ff:ff:ff
    inet 192.168.42.15/24 brd 192.168.42.255 scope global eth0
    inet6 fe80::250:bfff:fe7e:549a/10 scope link

benny@home benny> ip route show
192.168.42.0/24 dev eth0  scope link
127.0.0.0/8 dev lo  scope link
default via 192.168.42.1 dev eth0
```

### The ifconfig command

While **ip** is the most novel way to configure a Linux system, **ifconfig** is still very popular. Use it without option for displaying network interface information:

```
els@asus:~$ /sbin/ifconfig
eth0      Link encap:Ethernet  HWaddr 00:50:70:31:2C:14
          inet addr:60.138.67.31  Bcast:66.255.255.255  Mask:255.255.255.192
          inet6 addr: fe80::250:70ff:fe31:2c14/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:31977764 errors:0 dropped:0 overruns:0 frame:0
          TX packets:51896866 errors:0 dropped:0 overruns:0 carrier:0
          collisions:802207 txqueuelen:1000
          RX bytes:2806974916 (2.6 GiB)  TX bytes:2874632613 (2.6 GiB)
          Interrupt:11 Base address:0xec00

                                                                      lo
Link encap:Local Loopback
          inet addr:127.0.0.1  Mask:255.0.0.0
          inet6 addr: ::1/128 Scope:Host
          UP LOOPBACK RUNNING  MTU:16436  Metric:1
          RX packets:765762 errors:0 dropped:0 overruns:0 frame:0
          TX packets:765762 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:0
          RX bytes:624214573 (595.2 MiB)  TX bytes:624214573 (595.2 MiB)
```

Here, too, we note the most important aspects of the interface configuration:

- The IP address is marked with "inet addr".
- The hardware address follows the "HWaddr" tag.

Both **ifconfig** and **ip** display more detailed configuration information and a number of statistics about each interface and, maybe most important, whether it is "UP" and "RUNNING".



# Checking the host configuration with netstat

Apart from the **ip** command for displaying the network configuration, there's the common **netstat** command which has a lot of options and is generally useful on any UNIX system.

Routing information can be displayed with the `-nr` option to the **netstat** command:

```
bob:~> netstat -nr
Kernel IP routing table
Destination  Gateway      Genmask       Flags MSS Window irtt Iface
192.168.42.0 0.0.0.0      255.255.255.0 U      40 0          0 eth0
127.0.0.0    0.0.0.0      255.0.0.0     U      40 0          0 lo
0.0.0.0      192.168.42.1 0.0.0.0       UG     40 0          0 eth0
```

This is a typical client machine in an IP network. It only has one network device, *eth0*. The *lo* interface is the local loop.

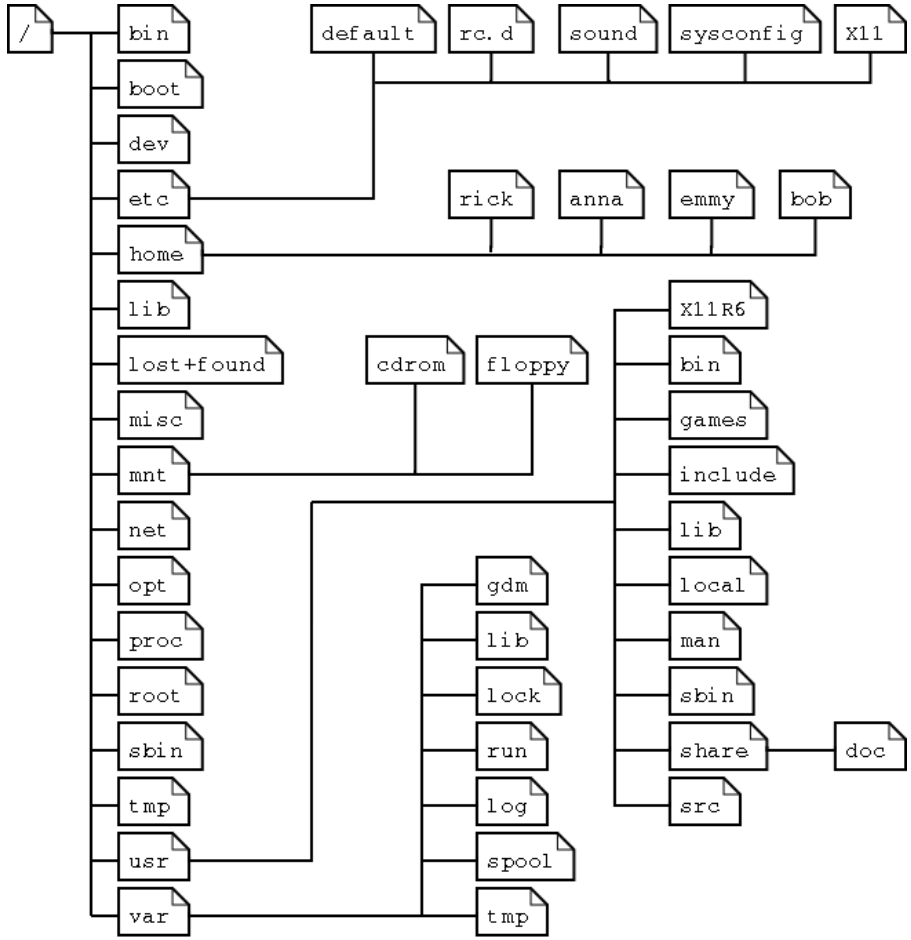
# 7. Basic User Commands

**Exit / logout**

For convenience, the Linux file system is usually thought of in a tree structure. On a standard Linux system you will find the layout generally follows the scheme presented below.



Figure Linux file system layout



**Subdirectories of the root directory**

| Directory | Content |
|---|---|
| /bin | Common programs, shared by the system, the system administrator and the users. |
| /boot | The startup files and the kernel, `vmlinuz`. In some recent distributions also `grub` data. Grub is the GRand Unified Boot loader and is an attempt to get rid of the many different boot-loaders we know today. |
| /dev | Contains references to all the CPU peripheral hardware, which are represented as files with special properties. |
| /etc | Most important system configuration files are in `/etc`, this directory contains data similar to those in the Control Panel in Windows |
| /home | Home directories of the common users. |
| /initrd | (on some distributions) Information for booting. Do not remove! |
| /lib | Library files, includes files for all kinds of programs needed by the system and the users. |
| /lost+found | Every partition has a `lost+found` in its upper directory. Files that were saved during failures are here. |
| /misc | For miscellaneous purposes. |
| /mnt | Standard mount point for external file systems, e.g. a CD-ROM or a digital camera. |
| /net | Standard mount point for entire remote file systems |
| /opt | Typically contains extra and third party software. |
| /proc | A virtual file system containing information about system resources. More information about the meaning of the files in `proc` is obtained by entering the command **man *proc*** in a terminal window. The file `proc.txt` discusses the virtual file system in detail. |
| /root | The administrative user's home directory. Mind the difference between /, the root directory and /root, the home directory of the *root* user. |
| /sbin | Programs for use by the system and the system administrator. |
| /tmp | Temporary space for use by the system, cleaned upon reboot, so don't use this for saving any work! |
| /usr | Programs, libraries, documentation etc. for all user-related programs. |
| /var | Storage for all variable files and temporary files created by users, such as log files, the mail queue, the print spooler area, space for temporary storage of files downloaded from the Internet, or to keep an image of a CD before burning it. |



## Absolute and relative paths

A path, which is the way you need to follow in the tree structure to reach a given file, can be described as starting from the trunk of the tree (the / or root directory). In that case, the path starts with a slash and is called an absolute path, since there can be no mistake: only one file on the system can comply.

In the other case, the path doesn't start with a slash and confusion is possible between `~/bin/wc` (in the user's home directory) and `bin/wc` in `/usr`, from the previous example. Paths that don't start with a slash are always relative.

## Creating directories

A way of keeping things in place is to give certain files specific default locations by creating directories and subdirectories (or folders and sub-folders if you wish). This is done with the **mkdir** command:

```
richard:~> mkdir archive

richard:~> ls -ld archive
drwxrwxrwx  2 richard richard          4096 Jan 13 14:09 archive/
```

Creating directories and subdirectories in one step is done using the -p option:

```
richard:~> cd archive

richard:~/archive> mkdir 1999 2000 2001

richard:~/archive> ls
1999/  2000/  2001/

richard:~/archive> mkdir 2001/reports/Restaurants-Michelin/
mkdir: cannot create directory `2001/reports/Restaurants-Michelin/':
No such file or directory

richard:~/archive> mkdir -p 2001/reports/Restaurants-Michelin/

richard:~/archive> ls 2001/reports/
Restaurants-Michelin/
```



## Moving files

Now that we have properly structured our home directory, it is time to clean up unclassified files using the **mv** command:

```
richard:~/archive> mv ../report[1-4].doc reports/Restaurants-Michelin/
```

## Copying files

Copying files and directories is done with the **cp** command. A useful option is recursive copy (copy all underlying files and subdirectories), using the -R option to **cp**. The general syntax is

```
cp [-R] fromfile tofile
```

## Removing files

Use the **rm** command to remove single files, **rmdir** to remove empty directories. (Use **ls -a** to check whether a directory is empty or not). The **rm** command also has options for removing non-empty directories with all their subdirectories, read the Info pages for these rather dangerous options.

## 8. Introduction to Shells:

# Environment variables

We already mentioned a couple of environment variables, such as PATH and HOME. Until now, we only saw examples in which they serve a certain purpose to the shell. But there are many other Linux utilities that need information about you in order to do a good job.

The environment variables are managed by the shell. As opposed to regular shell variables, environment variables are inherited by any program you start, including another shell. New processes are assigned a copy of these variables, which they can read, modify and pass on in turn to their own child processes.

There is nothing special about variable names, except that the common ones are in upper case characters by convention. You may come up with any name you want, although there are standard variables that are important enough to be the same on every Linux system, such as PATH and HOME.

### Exporting variables

An individual variable's content is usually displayed using the **echo** command, as in these examples:

```
debby:~> echo $PATH
/usr/bin:/usr/sbin:/bin:/sbin:/usr/X11R6/bin:/usr/local/bin

debby:~> echo $MANPATH
/usr/man:/usr/share/man/:/usr/local/man:/usr/X11R6/man
```



In Bash, we normally do this in one elegant step:

**export VARIABLE=*value***

The same technique is used for the MANPATH variable, that tells the **man** command where to look for compressed man pages. If new software is added to the system in new or unusual directories, the documentation for it will probably also be in an unusual directory. If you want to read the man pages for the new software, extend the MANPATH variable:

```
debby:~> export MANPATH=$MANPATH:/opt/FlightGear/man

debby:~> echo $MANPATH
/usr/man:/usr/share/man:/usr/local/man:/usr/X11R6/man:/opt/FlightGear/man
```

## Reserved variables

The following table gives an overview of the most common predefined variables:

**Table Common environment variables**

| Variable name | Stored information |
|---|---|
| DISPLAY | used by the X Window system to identify the display server |
| DOMAIN | domain name |
| EDITOR | stores your favorite line editor |
| HISTSIZE | size of the shell history file in number of lines |
| HOME | path to your home directory |
| HOSTNAME | local host name |
| INPUTRC | location of definition file for input devices such as keyboard |
| LANG | preferred language |
| LD_LIBRARY_PATH | paths to search for libraries |
| LOGNAME | login name |
| MAIL | location of your incoming mail folder |
| MANPATH | paths to search for man pages |
| OS | string describing the operating system |
| OSTYPE | more information about version etc. |
| PAGER | used by programs like **man** which need to know what to do in case output is more than one terminal window. |
| PATH | search paths for commands |
| PS1 | primary prompt |
| PS2 | secondary prompt |
| PWD | present working directory |
| SHELL | current shell |
| TERM | terminal type |
| UID | user ID |
| USER(NAME) | user name |



| Variable name | Stored information |
|---|---|
| VISUAL | your favorite full-screen editor |
| XENVIRONMENT | location of your personal settings for X behavior |
| XFILESEARCHPATH | paths to search for graphical libraries |

A lot of variables are not only predefined but also preset, using configuration files.

# 9. Fundamental Backup Techniques

Accidents will happen sooner or later. In this chapter, we'll discuss how to get data to a safe place

Upon completion of this chapter, you will know how to:

- Make, query and unpack file archives
- Handle floppy disks and make a boot disk for your system
- Write CD-ROMs
- Make incremental backups
- Create Java archives
- Find documentation to use other backup devices and programs
- Encrypt your data

## Archiving with tar

In most cases, we will first collect all the data to back up in a single archive file, which we will compress later on. The process of archiving involves concatenating all listed files and taking out unnecessary blanks. In Linux, this is commonly done with the **tar** command. **tar** was originally designed to archive data on tapes, but it can also make archives, known as *tarballs*.

**tar** has many options, the most important ones are cited below:

- `-v`: verbose
- `-t`: test, shows content of a tarball
- `-x`: extract archive
- `-c`: create archive
- `-f archivedevice`: use `archivedevice` as source/destination for the tarball, the device defaults to the first tape device (usually `/dev/st0` or something similar)



In the example below, an archive is created and unpacked.

```
gaby:~> ls images/
me+tux.jpg  nimf.jpg

gaby:~> tar cvf images-in-a-dir.tar images/
images/
images/nimf.jpg
images/me+tux.jpg

gaby:~> cd images

gaby:~/images> tar cvf images-without-a-dir.tar *.jpg
me+tux.jpg
nimf.jpg

gaby:~/images> cd

gaby:~> ls */*.tar
images/images-without-a-dir.tar

gaby:~> ls *.tar
images-in-a-dir.tar

gaby:~> tar xvf images-in-a-dir.tar
images/
images/nimf.jpg
images/me+tux.jpg

gaby:~> tar tvf images/images-without-dir.tar
-rw-r--r-- gaby/gaby  42888 1999-06-30 20:52:25 me+tux.jpg
-rw-r--r-- gaby/gaby   7578 2000-01-26 12:58:46 nimf.jpg

gaby:~> tar xvf images/images-without-a-dir.tar
me+tux.jpg
nimf.jpg
```

```
gaby:~> ls *.jpg
me+tux.jpg  nimf.jpg
```

## Compressing and unpacking with gzip or bzip2

Data, including tarballs, can be compressed using zip tools. The **gzip** command will add the suffix .gz to the file name and remove the original file.

```
jimmy:~> ls -la | grep tar
-rw-rw-r-- 1 jimmy  jimmy    61440 Jun  6 14:08 images-without-dir.tar

jimmy:~> gzip images-without-dir.tar

jimmy:~> ls -la images-without-dir.tar.gz
-rw-rw-r-- 1 jimmy  jimmy    50562 Jun  6 14:08 images-without-dir.tar.gz
```

Uncompress gzipped files with the `-d` option.

**bzip2** works in a similar way, but uses an improved compression algorithm, thus creating smaller files. See the **bzip2** info pages for more.

Linux software packages are often distributed in a gzipped tarball. The sensible thing to do after unpacking that kind of archives is find the README and read it. It will generally contain guidelines to installing the package.

The GNU **tar** command is aware of gzipped files. Use the command

**tar zxvf file.tar.gz**

for unzipping and untarring `.tar.gz` or `.tgz` files. Use

**tar jxvf file.tar.bz2**

for unpacking **tar** archives that were compressed with **bzip2**.



## Using the dd command to dump data

The **dd** command can be used to put data on a disk, or get it off again, depending on the given input and output devices. An example:

```
gaby:~> dd if=images-without-dir.tar.gz of=/dev/fd0H1440
98+1 records in
98+1 records out

gaby~> dd if=/dev/fd0H1440 of=/var/tmp/images.tar.gz
2880+0 records in
2880+0 records out

gaby:~> ls /var/tmp/images*
/var/tmp/images.tar.gz
```