Basic tools for the administrator

Josep Jorba Esteve

PID_00148464

UOC
Universitat Oberta de Catalunya
www.uoc.edu

Index

Introduction

On a daily basis, an administrator of GNU/Linux systems has to tackle a large number of tasks. In general, the UNIX philosophy does not have just one tool for every task or just one way of doing things. What is common is for UNIX systems to offer a large number of more or less simple tools to handle the different tasks.

It will be the combination of the basic tools, each with a well-defined task that will allow us to resolve a problem or administration task.

Note

GNU/Linux has a very broad range of tools with basic functionalities, whose strength lies in their combination.

In this unit we will look at different groups of tools, identify some of their basic functions and look at a few examples of their uses. We will start by examining some of the standards of the world of GNU/Linux, which will help us to find some of the basic characteristics that we expect of any GNU/Linux distribution. These standards, such as LSB (or Linux standard base) [Linc] and FHS (filesystem hierarchy standard) [Linb], tell us about the tools we can expect to find available, a common structure for the file system, and the various norms that need to be fulfilled for a distribution to be considered a GNU/Linux system and to maintain shared rules for compatibility between them.

For automating administration tasks we tend to use commands grouped into shell scripts (also known as command scripts), through language interpreted by the system's shell (command interpreter). In programming these shell scripts we are allowed to join the system's commands with flow control structures, and thus to have a fast prototype environment of tools for automating tasks.

Another common scheme is to use tools of compiling and debugging high level languages (for example C). In general, the administrator will use them to generate new developments of applications or tools, or to incorporate applications that come as source code and that need to be adapted and compiled.

We will also analyse the use of some graphics tools with regards to the usual command lines. These tools tend to facilitate the administrator's tasks but their use is limited because they are heavily dependent on the GNU/Linux distribution and version. Even so, there are some useful exportable tools between distributions.

Finally, we will analyse a set of essential tools for maintaining the system updated, the package management tools. The software served with the GNU/Linux distribution or subsequently incorporated is normally offered in units known as packages, which include the files of specific software, plus the vari-

ous steps required in order to prepare the installation and then to configure it or, where applicable, to update or uninstall specific software. And every distribution tends to carry management software for maintaining lists of installed or installable packages, as well as for controlling existing versions or various possibilities of updating them through different original sources.

1. Graphics tools and command line

There are a large number of tools, of which we will examine a small share in this and subsequent modules, which are provided as administration tools by third parties, independent from the distribution, or by the distributor of the GNU/Linux system itself.

These tools may cover more or fewer aspects of the administration of a specific task and can appear with various different interfaces: whether command line tools with various associated configuration options and/or files or text tools with some form of menus; or graphics tools, with more suitable interfaces for handling information, wizards to automate the tasks or web administration interfaces.

All of this offers us a broad range of possibilities where administration is concerned, but we will always have to evaluate the ease of using them with the benefits of using them, and the knowledge of the administrator responsible for these tasks.

The common tasks of a GNU/Linux administrator can include working with different distributions (for example, the ones we will discuss Fedora [Fed] or Debian [Debb] or any other) or even working with commercial variants of other UNIX systems. This entails having to establish a certain way of working that allows us to perform the tasks in the different systems in a uniform manner.

For this reason, throughout the different modules we will try to highlight the most common aspects and the administration techniques will be mostly performed at a low level through a command line and/or the editing of associated configuration files.

Any of the GNU/Linux distributions tends to include command line, text, or especially, graphics tools to complement the above and to a greater or lesser degree simplify task administration [Sm02]. But we need to take several things into account:

a) These tools are a more or less elaborate interface of the basic command line tools and corresponding configuration files.

b) Normally they do not offer all the features or configurations that can be carried out at a low level.

c) Errors may not be well managed or may simply provide messages of the type "this task could not be performed".

d) The use of these tools hides, sometimes completely, the internal functioning of the service or task. Having a good understanding of the internal functioning is basic for the administrator, especially if the administrator is responsible for correcting errors or optimising services.

e) These tools are useful for improving production once the administrator has the required knowledge to handle routine tasks more efficiently and to automate them.

f) Or, in the opposite case, the task may be so complex, require so many parameters or generate so much data, that it may become impossible to control it manually. In these cases, the high level tools can be very useful and make practicable tasks that are otherwise difficult to control. For example, this category would include visualisation tools, monitorisation tools, and summaries of tasks or complex services.

g) For automating tasks, these tools (of a higher level) may not be suitable: they may not have been designed for the steps that need taking or may perform them inefficiently. For example, a specific case would be creating users, where a visual tool can be very attractive because of the way of entering the data; but what if instead of entering one or a few users we want to enter a list of tens or hundreds of them? if not prepared for this, the tool will become totally inefficient.

h) Finally, administrators normally wish to personalise their tasks using the tools they find most convenient and easy to adapt. In this aspect, it is common to use basic low-level tools, and shell scripts (we will study the basics in this unit) combining them in order to form a task.

We may use these tools occasionally (or daily), if we have the required knowledge for dealing with errors that can arise or to facilitate a process that the tool was conceived for, but always controlling the tasks we implement and the underlying technical knowledge.

2. Standards

Standards, whether generic of UNIX or particular to GNU/Linux, allow us to follow a few basic criteria that guide us in learning how to execute a task and that offer us basic information for starting our job.

In GNU/Linux we can find standards, such as the FHS (*filesystem hierarchy standard*) [Linb], which tells us what we can find in the our system's file system structure (or where to look for it), or the LSB (*Linux standard base),* which discusses the different components that we tend to find in the systems [Linc].

Note

See FHS in: www.pathname.com/fhs

The **FHS** *filesystem hierchachy standard* describes the main file system tree structure (/), which specifies the structure of the directories and the main files that they will contain. This standard is also used to a greater or lesser extent for commercial UNIX, where originally there were many differences that made each manufacturer change the structure as they wished. The standard originally conceived for GNU/Linux was made to normalise this situation and avoid drastic changes. Even so, the standard is observed to varying degrees, most distributions follow a high percentage of the FHS, making minor changes or adding files or directories that did not exist in the standard.

Note

The FHS standard is a basic tool that allows us to understand the structure and functionality of the system's main file system.

A basic directories scheme could be:

- /bin: basic system utilities, normally programs used by users, whether from the system's basic commands (such as /bin/ls, list directory), shells (/bin/bash) etc.

- /boot: files needed for booting the system, such as the image of the Linux kernel, in /boot/vmlinuz.

- /dev: here we will find special files that represent the different possible devices in the system, access to peripherals in UNIX systems is made as if they were files. We can find files such as /dev/console, /dev/modem, /dev/mouse, /dev/cdrom, /dev/floppy... which tend to be links to more specific devices of the driver or interface type used by the devices: /dev/mouse, linked to /dev/psaux, representing a PS2 type mouse; or /dev/cdrom to /dev/hdc, a CD-ROM that is a device of the second IDE connector and master. Here we find IDE devices such as /dev/hdx, scsi /dev/sdx... with x varying according to the number of the device. Here we should mention that initially this directory was static, with the files predefined, and/or configured at specific moments, nowadays we use dynamic technology

techniques (such as hotplug or udev), that can detect devices and create /dev files dynamically when the system boots or while running, with the insertion of removable devices.

- /etc: configuration files. Most administration tasks will need to examine or modify the files contained in this directory. For example: /etc/passwd contains part of the information on the system's user accounts.

- /home: it contains user accounts, meaning the personal directories of each user.

- /lib: the system's libraries, shared by user programs, whether static (.a extension) or dynamic (.so extension). For example, the standard C library, in libc.so files or libc.a. Also in particular, we can usually find the dynamic modules of the Linux kernel, in /lib/modules.

- /mnt: point for mounting (mount command) file systems temporarily; for example: /mnt/cdrom, for mounting a disk in the CD-ROM reader temporarily.

- /media: for common mounting point of removable devices.

- /opt: the software added to the system after the installation is normally placed here; another valid installation is in /usr/local.

- /sbin: basic system utilities. They tend to be command reserved for the administrator (root). For example: /sbin/fsck to verify the status of the file systems.

- /tmp: temporary files of the applications or of the system itself. Although they are for temporary running, between two executions the application/service cannot assume that it will find the previous files.

- /usr: different elements installed on the system. Some more complete system software is installed here, in addition to multimedia accessories (icons, images, sounds, for example in: /usr/share) and the system documentation (/usr/share/doc). It also tends to be used in /usr/local for installing software.

- /var: log or status type files and/or error files of the system itself and of various both local and network services. For example, log files in /var/log, e-mail content in /var/spool/mail, or printing jobs in /var/spool/lpd.

These are some of the directories defined in the FHS for the root system, then for example it specifies some subdivisions, such as the content of /usr and /var, and the typical data and/or executable files expected to be found at minimum in the directories (see references to FHS documents).

Regarding the distributions, Fedora/Red Hat follows the FHS standard very closely. It only presents a few changes in the files present in /usr, /var. In /etc there tends to be a directory per configurable component and in /opt, /usr/local there is usually no software installed unless the user installs it. Debian follows the standard, although it adds some special configuration directories in /etc.

Another standard in progress is the LSB (*Linux standard base*) [Linc]. Its idea is to define compatibility levels between the applications, libraries and utilities, so that portability of applications is possible between distributions without too many problems. In addition to the standard, they offer test sets to check the compatibility level. LSB in itself is a collection of various standards applied to GNU/Linux.

Note

See standard specifications: http://www.linuxfoundation.org/en/Specifications

3. System documentation

One of the most important aspects of our administration tasks will be to have the right documentation for our system and installed software. There are numerous sources of information, but we should highlight the following:

a) man is by far the best choice of help. It allows us to consult the GNU/Linux manual, which is grouped into various sections corresponding to administration commands, file formats, user commands, C language calls etc. Normally, to obtain the associated help, we will have enough with:

```
man command
```

Every page usually describes the command together with its options and, normally, several examples of use. Sometimes, there may be more than one entry in the manual. For example, there may be a C call with the same name as a command; in this case, we would have to specify what section we want to look at:

```
man n command
```

with *n* being the section number.

There are also several tools for exploring the manuals, for example *xman* and *tkman*, which through a graphic interface help to examine the different sections and command indexes. Another interesting command is *apropos* word, which will allow us to locate man pages that discuss a specific topic (associated with the word).

b) info is another common help system. This program was developed by GNU to document many of its tools. It is basically a text tool where the chapters and pages can be looked up using a simple keyboard-based navigation system.

c) Applications documentation: in addition to certain man pages, it is common to include extra documentation in the applications, in the form of manuals, tutorials or simple user guides. Normally, these documentation components are installed in the directory /usr/share/doc (or /usr/doc depending on the distribution), where normally a directory is created for each application package (normally the application can have a separate documentation package).

d) Distributions' own systems. Red Hat tends to come with several CDs of consultation manuals that can be installed on the system and that come in HTML or PDF formats. Fedora has a documentation project on its webpage. Debian offers its manuals in the form of one more software package that is usually installed in /usr/doc. At the same time, it has tools that classify the documentation in the system, organising it by means of menus for visualisation, such as *dwww* or *dhelp*, which offer web interfaces for examining the system's documentation.

e) Finally, X desktops, such as Gnome and KDE, usually also carry their own documentation systems and manuals, in addition to information for developers, whether in the form of graphic help files in their applications or own applications that compile all the help files (for example devhelp in Gnome).

4. Shell scripting

The generic term shell is used to refer to a program that serves as an interface between the user and the GNU/Linux system's kernel. In this section, we will focus on the interactive text shells, which are what we will find as users once we have logged in the system.

> The shell is a system utility that allows users to interact with the kernel through the interpretation of commands that the user enters in the command line or files of the shell script type.

The shell is what the users see of the system. The rest of the operating system remains mostly hidden from them. The shell is written in the same way as a user process (program); it does not form part of the kernel, but rather is run like just another user program.

When our GNU/Linux system starts up, it tends to offer users an interface with a determined appearance; the interface may be a text or graphic interface. Depending on the modes (or levels) of booting the system, whether with the different text console modes or modes that give us a direct graphic start up in X Window.

In graphic start up modes, the interface consists of an access administrator to manage the user login procedure using a graphic cover page that asks for the corresponding information to be entered: user identification and passsword. Access managers are common in GNU/Linux: xdm (belonging to X Window), gdm (Gnome) and kdm (KDE), as well as a few others associated to different window managers. Once we have logged in, we will find ourselves in the X Window graphic interface with a windows manager such as Gnome or KDE. To interact through an interactive shell, all we will need to do is to open one of the available terminal emulation programs.

If our access is in console mode (text), once logged in, we will obtain direct access to the interactive shell.

Another case of obtaining an interactive shell is by remote access to the machine, whether through any of the text possibilities such as telnet, rlogin, ssh, or graphic possibilities such as the X Window emulators.

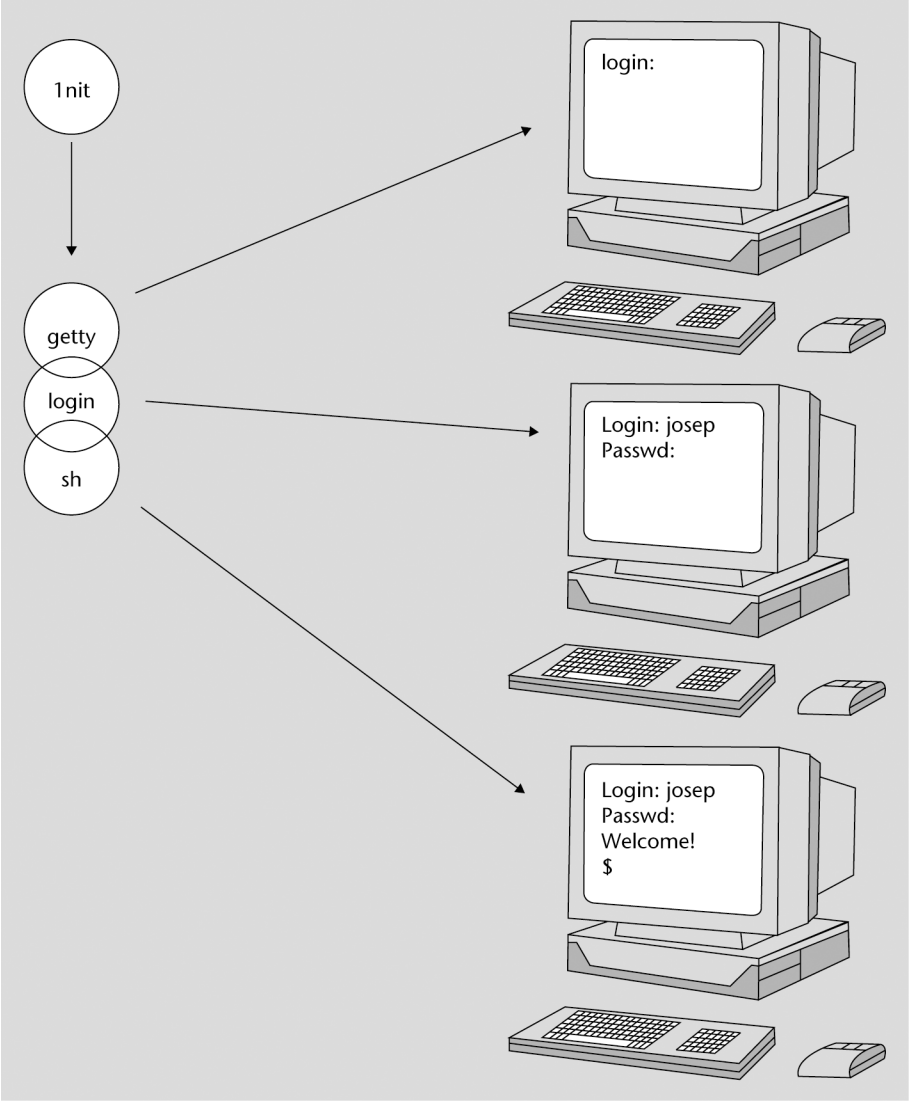

Figure 1. Example of starting up a text shell textual and the system processes involved [Oke]

4.1. Interactive *shells*

Having initiated the interactive shell [Qui01], the user is shown a prompt, indicating that a command line may be entered. After entering it, the shell becomes responsible for validating it and starting to run the required processes, in a number of phases:

- Reading and interpreting the command line.
- Evaluating wildcard characters such as $ * ? and others.
- Managing the required I/O redirections, pipes and background processes (&).
- Handling signals.
- Preparing to run programs.

Normally, command lines will be ways of running the system's commands, interactive shell commands, starting up applications or shell scripts.

Shell scripts are text files that contain command sequences of the system, plus a series of internal commands of the interactive shell, plus the necessary control structures for processing the program flow (of the type *while, for* etc.).

The system can run script files directly under the name given to the file. To run them, we invoke the shell together with the file name or we give the shell script execution permissions.

To some extent, we can see shell script as the code of an interpreted language that is executed on the corresponding interactive shell. For the administrator, shell scripts are very important, basically for two reasons:

1) The system's configuration and most of the services are provided through tools in the form of shell scripts.

2) The main way of automating administration processes is creating shell scripts.

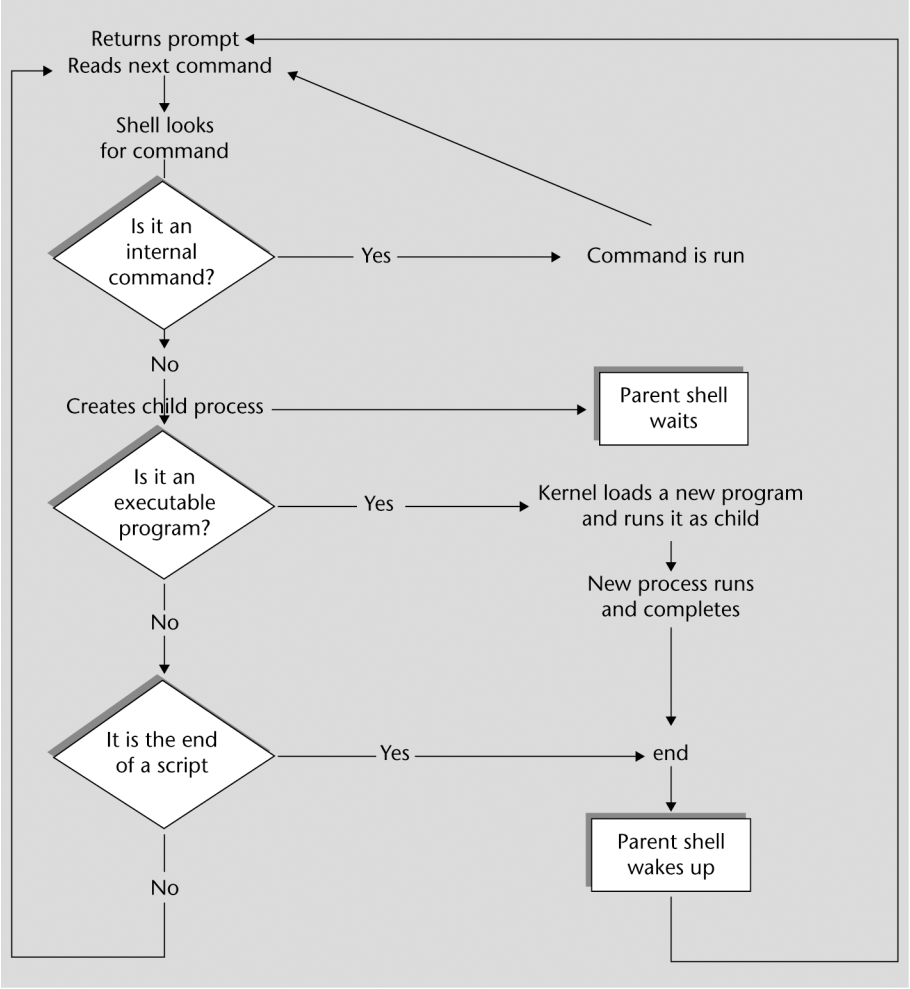

Figure 2. Basic shell flow control

All the programs that are invoked by a shell possess three predefined files, specified by the corresponding file handles. By default, these files are:

1) *standard input:* normally assigned to the terminal's keyboard (console); uses file handle number 0 (in UNIX the files use whole number file handles).

2) *standard output:* normally assigned to the terminal's screen; uses file handle 1.

3) *standard error:* normally assigned to the terminal's screen; uses file handle 2.

This tells us that any program run from the shell by default will have the input associated to the terminal's keyboard, the output associated to the screen, and that it will also send errors to the screen.

Also, the shells tend to provide the three following mechanisms:

1) Redirection: given that I/O devices and files are treated the same way in UNIX, the shell simply handles them all as files. From the user's point of view, the file handles can be reassigned so that the data flow of one file handle goes to any other file handle; this is called redirection. For example, we refer to redirecting file handles 0 or 1 as redirecting standard I/O.

2) Pipes: a program's standard output can be used as another's standard input by means of pipes. Various programs can be connected to each other using pipes to create what is called a pipeline.

3) Concurrence of user programs: users can run several programs simultaneously, indicating that they will be run in the background, or in the foreground, with exclusive control of the screen. Another way consists of allowing long jobs in the background while interacting with the shell and with other programs in the foreground.

In practice, in UNIX/Linux these shells entail:

- **Redirection**: a command will be able to receive input or output from other files or devices.

Example

let's see

```
command op file
```

where op may be:

- < : receive input from file.
- > : send output to file.
- >> : it indicates to add the output (by default, with > the file is created again).

- Pipes: chaining several commands, with transmission of their data:
  `command1 | command2 | command3`

- This instruction tells us that command1 will receive input possibly from the keyboard, send its output to command2, which will receive it as input and produce output towards command3, which will receive it and send its output to the standard output (by default, the screen).

- Background concurrence: any command executed with the '&' at the end of the line will be run in the background and the prompt of the shell will be returned immediately while it continues to be executed. We can follow the execution of commands with the *ps* command and its options, which allows us to observe the status of the system's processes. And we also have the *kill* order, which allows us to eliminate processes that are still being run or that have entered an error condition: *kill -9 PID* allows us to kill the process with PID identification number. PID is the identifier associated to the process, a whole number assigned to it by the system and that can be obtained using the *ps* command.

4.2. Shells

The shell's independence in relation to the operating system's kernel (the shell is just an interface layer), allows us to have several of them on the system [Qui01]. Although some of the more frequent ones are:

a) The Bash (initialism for *Bourne-again shell*). The default GNU/Linux shell.

b) The Bourne shell (sh). This has always been the standard UNIX shell, and the one that all UNIX systems have in some version. Normally, it is the administrator's default shell (root). In GNU/Linux it tends to be Bash, an improved version of the Bourne shell, which was created by Stephen Bourne at AT&T at the end of the seventies. The default prompt tends to be a '$' (in root a '#').

c) The Korn shell (ksh). It is a supergroup of Bourne (some compatibility is maintained), written at AT&T by David Korn (in the mid eighties), which some functionalities of Bourne and C, with some additions. The default prompt is the $.

d) The C shell (csh). It was developed at the University of Berkeley by Bill Joy towards the end of the seventies and has a few interesting additions to Bourne, like a command log, alias, arithmetic from the command line, it completes file names and controls jobs in the background. The default prompt for users is '%'. UNIX users tend to prefer this shell for interaction, but UNIX administrators prefer to use Bourne, because the scripts tend to be more compact and to execute faster. At the same time, an advantage of the scripts in C shell is that, as the name indicates, the syntax is based on C language (although it is not the same).

e) Others, such as restricted or specialised versions of the above.

The Bash (*Bourne again shell*) [Bas] [Coo] has grown in importance since it was included in GNU/Linux systems as the default shell. This shell forms part of the GNU software project. It is an attempt to combine the three preceding shells (Bourne, C and Korn), maintaining the syntax of the original Bourne shell. This is the one we will focus on in our subsequent examples.

A rapid way of knowing what shell we are in as users is by using the variable $*SHELL*, from a command line with the instruction:

```
echo $ SHELL
```

We will find that some aspects are common to all shells:

- They all allow shell scripts to be written, which are then interpreted executing them either by the name (if the file has an execution permission) or by passing it as a parameter to the command of the shell.

- System users have a default shell associated to them. This information is provided upon creating the users' accounts. The administrator will assign a shell to each user, or otherwise the default shell will be assigned (*bash* in GNU/Linux). This information is saved in the passwords file in /etc/passwd and can be changed with the *chsh* command, this same command with the option -l will list the system's available shells (see also */etc/shells*).

- Every shell is actually an executable command, normally present in the /bin directories in GNU/Linux (or /usr/bin).

• Shell scripts can be written in any of them, but adjusting to each one's syntax, which is normally different (sometimes the differences are minor). The construction syntax, as well as the internal commands, are documented in every shell's man page (*man bash* for example).

• Every shell has some associated start up files (initialisation files), and every user can adjust them to their needs, including code, variables, paths...

• The capacity in the programming lies in the combination of each shell's syntax (of its constructions), with the internal commands of each shell, and a series of UNIX commands that are commonly used in the scripts, like for example *cut*, *sort*, *cat*, *more*, *echo*, *grep*, *wc*, *awk*, *sed*, *mv*, *ls*, *cp*...

Note

To program a shell it is advisable to have a good knowledge of these UNIX commands and of their different options.

• If as users we are using a specific shell, nothing prevents us from starting up a new copy of the shell (we call it a subshell), whether it is the same one or a different one. We simply invoke it through the name of the executable, whether *sh*, *bash*, *csh* or *ksh*. Also when we run a shell script a subshell is launched with the corresponding shell for executing the requested script.

Some basic differences between them [Qui01]:

a) Bash is the default shell in GNU/Linux (unless otherwise specified in creating the user account). In other UNIX systems it tends to be the Bourne shell (sh). Bash is compatible with sh and also incorporates some features of the other shells, csh and ksh.

b) Start-up files: sh, ksh have *.profile* (in the user account, and is executed in the user's login) and ksh also tends to have a *.kshrc* which is executed next, csh uses *.login* (it is run when the user login initiates one time only), *.logout* (before leaving the user's session) and *.cshrc* (similar to the *.profile,* in each initiated C subshell). And Bash uses the *.bashrc* and the *.bash_profile*. Also, the administrator can place common variables and paths in the */etc/profile* file that will be executed before the files that each user has. The shell start-up files are placed in the user's account when it is created (normally they are copied from the */etc/skel* directory), where the administrator can leave some skeletons of the prepared files.

c) The system or service configuration scripts are usually written in Bourne shell (sh), since most UNIX systems used them this way. In GNU/Linux we can also find some in Bash and also in other script languages not associated to the shell such as Perl or Python.

d) We can identify what shell the script is run on using the *file* command, for example *file <scriptname>*. Or by examining the first line of the script, which tends to be: *#!/bin/name*, where the name is *bash, sh, csh, ksh*... This line tells us, at the moment of running the script, what shell needs to be used to interpret it (in other words, what subshell needs to be launched in order to run it). It is important for all scripts to contain it, since otherwise they will try to run the default shell (Bash in our case) and the syntax may not be the right one, causing many syntax errors in the execution.

4.3. System variables

Some useful system variables (we can see them using the *echo* command for example), which can be consulted in the command line or within the programming of the shell scripts are:

Variable	Value Example	Description
HOME	/home/juan	Root directory of the user
LOGNAME	juan	User ID at *login*
PATH	/bin:/usr/local/bin:/usr/X11/bin	Paths
SHELL	/bin/bash	User *shell*
PS1	$	*Shell* prompt, the user can change it
MAIL	/var/mail/juan	E-mail directory
TERM	xterm	Type of terminal used by the user
PWD	/home/juan	Current user directory

The different variables of the environment can be seen using the *env* command. For example:

```
$ env
SSH_AGENT_PID = 598
MM_CHARSET = ISO-8859-15
TERM = xterm
DESKTOP_STARTUP_ID =
SHELL = /bin/bash
WINDOWID = 20975847
LC_ALL = es_ES@euro
USER = juan
LS_COLORS = no = 00:fi = 00:di = 01;34:ln = 01;
SSH_AUTH_SOCK = /tmp/ssh-wJzVY570/agent.570
SESSION_MANAGER = local/aopcjj:/tmp/.ICE-unix/570
USERNAME = juan
PATH=/soft/jdk/bin:/usr/local/bin:/usr/bin:/bin:/usr/bin/
X11:/usr/games
```

```
MAIL = /var/mail/juan
PWD = /etc/skel
JAVA_HOME = /soft/jdk
LANG = es_ES@euro
GDMSESSION = Gnome
JDK_HOME = /soft/jdk
SHLVL = 1
HOME = /home/juan
GNOME_DESKTOP_SESSION_ID = Default
LOGNAME = juan
DISPLAY = :0.0
COLORTERM = gnome-terminal
XAUTHORITY = /home/juan/.Xauthority
_ = /usr/bin/env
OLDPWD = /etc
```

4.4. Programming scripts in Bash

Here we will look at some basic concepts of the shell scripts in Bash, we advise further reading in [Bas] [Coo].

All Bash scripts have to start with the line:

```
#!/bin/bash
```

This line indicates the shell used by the user, the one active at the time, what shell is needed for running the script that appears next.

The script can be run in two different ways:

1) By running directly from the command line, on condition it has an execution permission. If this is not the case, we can establish the permission with: *chmod +x script*.

2) By running through the shell, we call on the shell explicitly: */bin/bash script*.

We should take into account that, irrespective of the method of execution, we are always creating a subshell where our script will be run.

4.4.1. Variables in Bash

The assignment of variables is done by:

```
variable = value
```

The value of the variable can be seen with:

```
echo $variable
```

where '$' refers us to the variable's value.

The default variable is only visible in the script (or in the shell). If the variable needs to be visible outside the script, at the level of the shell or any subshell that is generated a posteriori, we will need to "export" it as well as assign it. We can do two things:

- Assign first and export after:

```
var=value
export var
```

- Export during assignment:

```
export var=value
```

In Bash scripts we have some accessible predetermined variables:

- $1-$N: It saves past arguments as parameters to the script from the command line.
- $0: It saves the script name, it would be parameter 0 of the command line.
- $*: It saves all parameters from 1 to N of this variable.
- $: It saves both parameters, but with double inverted commas (" ") for each of them.
- $?: "Status": it saves the value returned by the most recent executed command. Useful for checking error conditions, since UNIX tends to return 0 if the execution was correct, and a different value as an error code.

Another important issue regarding assignments is the use of inverted commas:

- Double inverted commas allow everything to be considered as a unit.
- Single inverted commas are similar, but ignore the special characters inside them.
- Those pointed to the left (`command`) are used for evaluating the inside, if there is an execution or replacement to be made. First the content is executed, and then what there was is replaced by the result of the execution. For example: var = 'ls' saves the list of the directory in $var.

4.4.2. Comparisons

For conditions the order *test expression* tends to be used or directly *[expression]*. We can group available conditions in:

- Numerical comparison: -eq, -ge, -gt, -le, -lt, -ne, corresponding to: equal to, greater than or equal to (ge), greater than, less than or equal to (le), less than, not equal to.

- Chain comparison: :=, !=, -n, -z, corresponding to chains of characters: equal, different, with a greater length than 0, length equal to zero or empty.

- File comparison: -d, -f -r, -s, -w, -x. The file is: a directory, an ordinary file, is readable, is not empty, is writable, is runnable.

- Booleans between expressions: !, -a, -o, conditions of not, and, and or.

4.4.3. Control structures

Regarding the script's internal programming, we need to think that we are basically going to find:

- Commands of the operating system itself.
- Internal commands of the Bash (see: man bash).
- Programming control structures (for, while...), with the syntax of Bash.

The basic syntax of control structures is as follows:

a) Structure *if...then,* evaluates the expression and if a certain value is obtained, then the commands are executed.

```
if [ expresion ]
 then
  commands
fi
```

b) Structure if...then...else, evaluates the expression and if a certain value is obtained then the commands1 are executed, otherwise comands2 are executed:

```
if [ expresion ]
 then
  commands1
 else
```

```
  commands2
fi
```

c) Structure if..then...else if...else, same as above, with additional if structures.

```
if [ expresion ]
 then
  commands
 elif [ expresion2 ]
 then
  commands
 else
  commands
fi
```

d) Structure *case select*, multiple selection structure according to the selection value (in *case*)

Note

Shells such as Bash offer a wide set of control structures that make them comparable to any other language.

```
case string1 in
 str1)
  commands;;
 str2)
  commands;;
 *)
  commands;;
esac
```

e) Loop *for*, replacement of the variable for each element of the list:

```
for var1 in list
do
  commands
done
```

f) Loop *while*, while the expression is fulfilled:

```
while [ expresion ]
do
  commands
done
```

g) Loop *until*, until the expression is fulfilled:

```
until [ expression ]
do
  commands
```

```
done
```

h) Declaration of functions:

```
fname() {
  commands
}
```

or with a call accompanied by parameters:

```
fname2(arg1,arg2...argN) {
  commands
}
```

and function calls with fname or fname2 p1 p2 p3 ... pN.

5. Package management tools

In any distribution, the packages are the basic item for handling the tasks of installing new software, updating existing software or eliminating unused software.

Basically, a package is a set of files that form an application or the combination of several related applications, normally forming a single file (known as a package), with its own format, normally compressed, which is distributed via CD/DVD or downloading service (ftp or http repositories).

The use of packages is helpful for adding or removing software, because it considers it as a unit instead of having to work with the individual files.

In the distribution's content (its CD/DVDs) the packages tend to be grouped into categories such as: a) base: essential packages for the system's functioning (tools, start-up programs, system libraries); b) system: administration tools, utility commands; c) development: programming tools: editors, compilers, debuggers... d) graphics: graphics controllers and interfaces, desktops, windows managers... e) other categories.

Normally, to install a package we will need to follow a series of steps:

1) Preliminary steps (pre-installation): check that the required software exists (and with the correct versions) for its functioning (dependencies), whether system libraries or other applications used by the software.

2) Decompress the package content, copying the files to their definitive locations, whether absolute (with a fixed position) or can be relocated to other directories.

3) Post-installation: retouching the necessary files, configuring possible software parameters, adjusting it to the system...

Depending on the types of packages, these steps may be mostly automatic (this is the case in RPM [Bai03] and DEB [Deb02]) or they may all be needed to be done by hand (.tgz case) depending on the tools provided by the distribution.

Next, let's see perhaps the three most classical packages of most distributions. Each distribution has one as standard and supports one of the others.

5.1. TGZ package

TGZ packages are perhaps those that have been used for longest. The first GNU/Linux distributions used them for installing the software, and several distributions still use it (for example, Slackware) and some commercial UNIX. They are a combination of files joined by the tar command in a single .tar file that has then been compressed using the gzip utility, and that tends to appear with the .tgz or .tar.gz extension. At the same time, nowadays it is common to find tar.bz2 which instead of gzip use another utility called bzip2, which in some cases obtains greater file compression.

Note

TGZ packages are a basic tool when it comes to installing unorganised software. Besides, they are a useful tool for back-up processes and restoring files.

Contrary to what it may seem, it is a commonly used format especially by the creators or distributors of software external to the distribution. Many software creators that work for various platforms, such as various commercial UNIX and different distributions of GNU/Linux prefer it as a simpler and more portable system.

An example of this case is the GNU project, which distributes its software in this format (in the form of source code), since it can be used in any UNIX, whether a proprietary system, a BSD variant or a GNU/Linux distribution.

If in binary format, we will have to bear in mind that it is suitable for our system, for example a denomination such as the following one is common (in this case, version 1.4 of the Mozilla web navigator):

```
mozilla-i686-pc-linux-gnu-1.4-installer.tar.gz
```

where we have the package name, as Mozilla, designed for i686 architecture (Pentium II or above or compatible), it could be i386, i586, i686, k6 (amd k6), k7 (amd athlon), amd64 u x86_64 (for AMD64 and some 64bit intels with em64t), o ia64 (intel Itaniums) others for the architectures of other machines such as sparc, powerpc, mips, hppa, alpha... then it tells us that it is for Linux, on a PC machine, software version 1.4.

If it were in source format, it could appear as:

```
mozilla-source-1.4.tar.gz
```

where we are shown the word *source*; in this case it does not mention the machine's architecture version, this tells us that it is ready for compiling on different architectures.

Otherwise, there would be different codes for every operating system or source: GNU/Linux, Solaris, Irix, bsd...

The basic process with these packages consists of:

1) Decompressing the package (they do not tend to use absolute path, meaning that they can be decompressed anywhere):

```
tar -zxvf file.tar.gz (or .tgz file)
```

With the *tar* command we use z options: decompress, x: extract files, v: view process, f: name the file to be treated.

It can also be done separately (without the tar's z):

```
gunzip file.tar.gz
```

(leaves us with a tar file)

```
tar -xvf file.tar
```

2) Once we have decompressed the tgz, we will have the files it contained, normally the software should include some file of the *readme* or *install* type, which specifies the installation options step by step, and also possible software dependencies.

In the first place, we should check the dependencies to see if we have the right software, and if not, look for it and install it.

If it is a binary package, the installation is usually quite easy, since it will be either directly executable from wherever we have left it or it will carry its own installer. Another possibility is that we may have to do it manually, meaning that it will be enough to copy it (cp -r, recursive copy) or to move it (*mv* command) to the desired position.

Another case is the source code format. Then, before installing the software we will first have to do a compilation. For this we will need to read the instruction that the program carries in some detail. But most developers use a GNU system called autoconf (from autoconfiguration), which normally uses the following steps (if no errors appear):

- ./*configure*: it is a script that configures the code so that it can be compiled on our machine and that verifies that the right tools exist. The --prefix = directory option makes it possible to specify where the software will be installed.

- *make*: compilation itself.

- *make install*: installing the software in the right place, normally previously specified as an option to configure or assumed by default.

This is a general process, but it depends on the software whether it follows it or not, there are fairly worse cases where the entire process needs to be carried out by hand, retouching configuration files or the *makefile*, and/or compiling the files one by one, but luckily this is becoming less and less common.

In the case of wanting to delete all of the installed software, we will have to use the uninstaller if provided or, otherwise, directly delete the directory or installed files, looking out for potential dependencies.

The tgz packages are fairly common as a backup mechanism for administration tasks, for example, for saving copies of important data, making backups of user accounts or saving old copies of data that we do not know if we will need again. The following process tends to be used: let's suppose that we want to save a copy of the directory "dir", we can type: `tar -cvf dir.tar dir` (c: compact dir in the file dir.tar) `gzip dir.tar` (compress) or in a single instruction like:

```
tar -cvzf dir.tgz dir
```

The result will be a dir.tgz file. We need to be careful if we are interested in conserving the file attributes and user permissions, as well as possibly links that may exist (we must examine the tar options so that it adjusts to the required backup options).

5.2. Fedora/Red Hat: RPM packages

The RPM packages system [Bai03] created by Red Hat represents a step forward, since it includes the management of software configuration tasks and dependencies. Also, the system stores a small database with the already installed packages, which can be consulted and updated with new installations.

Conventionally, RPM packages use a name such as:

```
package-version-rev.arq.rpm
```

where *package* is the name of the software, *version* is the numbering of the software version, *rev* normally indicates the revision of the RPM package, which indicates the number of times it has been built and *arq* refers to the architecture that it is designed for, whether Intel/AMD (i386, i586, i686, x86_64, em64t, ia64) or others such as alpha, sparc, PPC... The noarch "architecture" is normally used when it is independent, for example, a Set of scripts and src in the case of dealing with source code packages. A typical execution will include running rpm, the options of the operation to be performed, together with one or more names of packages to be processed together.

Note

The package: apache-1.3.19-23.i686.rpm would indicate that it is Apache software (the web server), in its version 1.3.19, package revision RPM 23, for Pentium II architectures or above.

Typical operations with RPM packages include:

- **Package information**: specific information about the package is consulted using the option -q together with the package name (with -p if on an rpm file). If the package has not yet been installed, the option would be -q accompanied by the information option to be requested, and if the request is to be made to all the installed packages at the same time, the option would be -qa. For example, requests from an installed package:

Request	RPM options	Results
Files	`rpm -ql`	List of the files it contains
Information	`rpm -qi`	Package description
Requirements	`rpm -qR`	Prior requirements, libraries or software

- **Installation**: simply `rpm -i package.rpm`, or with the URL where the package can be found, for downloading from FTP or web servers, we just need to use the syntax ftp:// or http:// to obtain the package's location.
  The installation can be completed on condition that the package dependencies are met, whether in the form of prior software or the libraries that should be installed. In the case of not fulfilling this requirement, we will be told what software is missing, and the name of the package that provides it. We can force the installation (although the installed software may not work) with the options `--force` or `--nodeps`, or simply by ignoring the information on the dependencies.
  The task of installing a package (done by `rpm`) entails various sub-tasks: a) checking for potential dependencies; b) examining for conflicts with other previously installed packages; c) performing pre-installation tasks; c) deciding what to do with the configuration files associated to the package if they existed previously; d) unpackaging the files and placing them in the right place; e) performing other post-installation tasks; finally, f) storing the log of tasks done in the RPM database.

- **Updating**: equivalent to the installation but first checking that the software already exists `rpm -U package.rpm`. It will take care of deleting the previous installation.

- **Verification**: during the system's normal functioning many of the installed files will change. In this regard, RPM allows us to check files in order to detect any changes from a normal process or from a potential error that could indicate corrupt data. Through `rpm -V package` we verify a specific package and through `rpm -Va` we will verify all of them.

- **Deletion**: erasing the package from the RPM system (-e or --erase); if there are dependencies, we may need to eliminate others previously.

Example

For a remote case:

rpm -i ftp://site/directory/package.rpm

would allow us to download the package from the provided FTP or web site, with its directory location, and proceed in this case to install the package.

We need to control where the packages come from and only use known and reliable package sources, such as the distribution's own manufacturer or trustworthy sites. Normally, together with the packages, we are offered a digital signature for them, so that we can check their authenticity. The sums md5 are normally used for checking that the package has not been altered and other systems, such as GPG (GNU version of PGP), for checking the authenticity of the package issuer. Similarly, we can find different RPM package stores on Internet, where they are available for different distributions that use or allow the RPM format.

Note

View the site:
www.rpmfind.net.

For a secure use of the packages, official and some third party repositories currently sign the packages electronically, for example, using the abovementioned GPG; this helps us to make sure (if we have the signatures) that the packages come from a reliable source. Normally, every provider (the repository) will include some PGP signature files with the key for its site. From official repositories they are normally already installed, if they come from third parties we will need to obtain the key file and include it in RPM, typically:

```
$ rpm –import GPG-KEY-FILE
```

With GPP-KEY-FILE being the GPG key file or URL of the file, normally this file will also have sum md5 to check its integrity. And we can find the keys in the system with:

```
$ rpm -qa | grep ^gpg-pubkey
```

we can observe more details on the basis of the obtained key:

```
$ rpm -qi gpg-key-xxxxx-yyyyy
```

For a specific RPM package we will be able to check whether it has a signature and with which one it has been used:

```
$ rpm –checksig -v <package>.rpm
```

And to check that a package is correct based on the available signatures, we can use:

```
$ rpm -K <package.rpm>
```

We need to be careful to import just the keys from the sites that we trust. When RPM finds packages with a signature that we do not have on our system or when the package is not signed, it will tell us and, then, we will have to decide on what we do.

Regarding RPM support in the distributions, in Fedora (Red Hat and also in its derivatives), RPM is the default package format and the one used extensively by the distribution for updates and software installation. Debian uses the format called DEB (as we will see), there is support for RPM (the rpm command exists), but only for consulting or package information. If it is essential to install an rpm package in Debian, we advise using the *alien* utility, which can convert package formats, in this case from RPM to DEB, and proceed to install with the converted package.

In addition to the distribution's basic packaging system, nowadays each one tends to support an intermediate higher level software management system, which adds an upper layer to the basic system, helping with software management tasks, and adding a number of utilities to improve control of the process.

In the case of Fedora (Red Hat and derivatives) it uses the YUM system, which allows as a higher level tool to install and manage packages in rpm systems, as well as automatic management of dependencies between packages. It allows access to various different repositories, centralises their configuration in a file (/etc/yum.conf normally), and has a simple commands interface.

Note

YUM in: http://yum.baseurl.org

The yum configuration is based on:

/etc/yum.config	(options file)
/etc/yum	(directory for some associated utilities)
/etc/yum.repos.d	(directory for specifying repositories, a file for each one, including access information and location of the gpg signatures).

A summary of the typical yum operations would be:

Order	Description
yum install <name>	Install the package with the name
yum update <name>	Update an existing package
yum remove <name>	Eliminate package
yum list <name>	Search package by name (name only)
yum search <name>	More extensive search
yum provices <file>	Search for packages that provide the file
yum update	Update the entire system
yum upgrade	As above, including additional packages

Finally, Fedora also offers a couple of graphics utilities for YUM, pup for controlling recently available updates, and pirutas a software management package. There are also others like *yumex*, with greater control of yum's internal configuration.

5.3. Debian: DEB packages

Debian has interactive tools such as *tasksel*, which makes it possible to select sub-sets of packages grouped into types of tasks: packages for X, for development, for documentation etc., or such as *dselect*, which allows us to navigate the entire list of available packages (there are thousands) and select those we wish to install or uninstall. In fact, these are only a front-end of the APT mid-level software manager.

At the command line level it has *dpkg*, which is the lowest level command (would be the equivalent to rpm), for managing the DEB software packages directly [Deb02], typically *dpkg -i package.deb* to perform the installation. All sorts of tasks related to information, installation, removal or making internal changes to the software packages can be performed.

The intermediary level (as in the case of Yum in Fedora) is presented by the APT tools (most are *apt-xxx* commands). APT allows us to manage the packages from a list of current and available packages based on various software sources, whether the installation's own CDs, FTP or web (HTTP) sites. This management is conducted transparently, in such a way that the system is independent from the software sources.

The APT system is configured from the files available in `/etc/apt`, where `/etc/apt/sources.list` is the list of available sources; an example could be:

```
deb http://http.us.debian.org/debian stable main contrib non-
free
```

```
debsrc http://http.us.debian.org/debian stable main contrib
non-free
deb http://security.debian.org stable/updates main contrib
non-free
```

Where various of the "official" sources for a Debian are compiled (*etch* in this case, which is assumed to be stable), from which we can obtain the software packages in addition to their available updates. Basically, we specify the type of source (web/FTP in this case), the site, the version of the distribution (stable in this example) and categories of software to be searched for (free, third party contributions, non-free or commercial licenses).

The software packages are available for the different versions of the Debian distribution, there are packages for the stable, testing, and unstable versions. The use of one or the others determines the type of distribution (after changing the repository sources in sources.list). It is possible to have mixed package sources, but it is not advisable, because conflicts could arise between the versions of the different distributions.

Note

Debian's DEB packages are perhaps the most powerful installation system existing in GNU/Linux. A significant benefit is the system's independence from the sources of the packages (through APT).

Once we have configured the software sources, the main tool for handling them in our system is apt-get, which allows us to install, update or remove from the individual package, until the entire distribution is updated. There is also a front-end to apt-get, called *aptitude*, whose options interface is practically identical (in fact it could be described as an apt-get emulator, since the interface is equivalent); as benefits it manages package dependencies better and allows an interactive interface. In fact it is hoped that aptitude will become the default interface in the command line for package management in Debian.

Some basic functions of apt-get:

- Installation of a particular package:
  `apt-get install package`

- Removing a package:
  `apt-get remove package`

- Updating the list of available packages:
  `apt-get update`

- Updating the distribution, we could carry out the combined steps:
  ```
  apt-get update
  apt-get upgrade
  apt-get dist-upgrade
  ```

Through this last process, we can keep our distribution permanently updated, updating installed packages and verifying dependencies with the new ones. Some useful tools for building this list are `apt-spy`, which tries to search for

the fastest official sites, or netselect, which allows us to test a list of sites. On a separate note, we can search the official sites (we can configure these with apt-setup) or copy an available source file. Additional (third party) software may need to add more other sources (to *etc/sources.list*); lists of available source sites can be obtained (for example: http://www.apt-get.org).

Updating a system in particular generates a download of a large number of packages (especially in `unstable`), which makes it advisable to empty the cache, the local repository, with the downloaded packages (they are kept in /var/cache/apt/archive) that will no longer be used, either with `apt-get clean` to eliminate them all or with `apt-get autoclean` to eliminate the packages that are not required because there are already new versions and, in principle, they will no longer be needed. We need to consider whether we may need these packages again for the purposes of reinstalling them, since, if so, we will have to download them again.

The APT system also allows what is known as SecureAPT, which is the secure management of packages through verifying sums (md5) and the signatures of package sources (of the GPG type). If the signatures are not available during the download, apt-get reports this and generates a list of unsigned packages, asking whether they will stop being installed or not, leaving the decision to the administrator. The list of current reliable sources is obtained using:

```
# apt-key list
```

The GPG keys of the official Debian sites are distributed through a package, and we can install them as follows:

```
# apt-get install debian-archive-keyring
```

Obviously, considering that we have the sources.list with the official sites. It is hoped that by default (depending on the version of Debian) these keys will already be installed when the system initiates. For other unofficial sites that do not provide the key in a package, but that we consider trustworthy, we can import their key, obtaining it from the repository (we will have to consult where the key is available, there is no defined standard, although it is usually on the repository's home page). Using apt-key add with the file, to add the key or also:

```
# gpg –import file.key
# gpg –export –armor XXXXXXXX | apt-key add -
```

With X being a hexadecimal related to the key (see repository instructions for the recommended way of importing the key and the necessary data).

Another important functionality of the APT system is for consulting package information, using the apt-cache tool, which allows us to interact with the lists of Debian software packages.

Example

The apt-cache tool has commands that allow us to search for information about the packages, for example:

- Search packages based on an incomplete name:
  `apt-cache search name`
- Show package description:
  `apt-cache show package`
- What packages it depends on:
  `apt-cache depends package`

Other interesting apt tools or functionalities:

- apt-show-versions: tells us what packages may be updated (and for what versions, see option -u).

Other more specific tasks will need to be done with the lowest level tool, such as dpkg. For example, obtaining the list of files of a specific installed package:

```
dpkg -L package
```

The full list of packages with

```
dpkg -l
```

Or searching for what package an element comes from (file for example):

```
dpkg -S file
```

This functions for installed packages; apt-file can also search for packages that are not yet installed.

Finally, some graphic tools for APT, such as synaptic, gnome-apt for gnome, and kpackage or adept for KDE are also worth mentioning, as well as the already mentioned text ones such as aptitude or dselect.

Conclusion: we should highlight that the APT management system (in combination with the dpkg base) is very flexible and powerful when it comes to managing updates and is the package management system used by Debian and its derived distributions such as Ubuntu, Kubuntu, Knoppix, Linex etc.

6. Generic administration tools

In the field of administration, we could also consider some tools, such as those designed generically for administration purposes. Although it is difficult to keep up to date with these tools because of the current plans of distribution with different versions, which evolve very quickly. We will mention a few examples (although at a certain time they may not be completely functional):

a) Linuxconf: this is a generic administration tool that groups together different aspects in a kind of text menu interface, which in the latest versions evolved to web support; it can be used with practically any GNU/Linux distribution and supports various details inherent to each one (unfortunately, it has not been updated for a while).

Note

We can find them in: Linuxconf http://www.solucorp.qc.ca/linuxconf

b) Webmin: this is another administration tool conceived from a web interface; it functions with a series of plug-ins that can be added for each service that needs to be administered; normally it has forms that specify the service configuration parameters; it also offers the possibility (if activated) of allowing remote administration from any machine with a navigator.

c) Others under development like cPanel, ISPConfig.

At the same time, the Gnome and KDE desktop environments tend to include the "Control Panel" concept, which allows management of the graphical interfaces' visual aspect as well as the parameters of some system devices.

With regards to the individual graphics tools for administration, the GNU/Linux distribution offers some directly (tools that accompany both Gnome and KDE), tools dedicated to managing a device (printers, sound, network card etc.), and others for the execution of specific tasks (Internet connection, configuring the start up of system services, configuring X Window, visualising logs...). Many of them are simple front-ends for the system's basic tools, or are adapted to special features of the distribution.

In this section, we should particularly highlight the Fedora distribution (Red Hat and derivatives), which tries to offer several (rather minimalist) utilities for different administration functions, we can find them on the desktop (in the administration menu), or in commands like system-config-xxxxx for different management functionalities for: screen, printer, network, security, users, packages etc. We can see some of them in the figure:

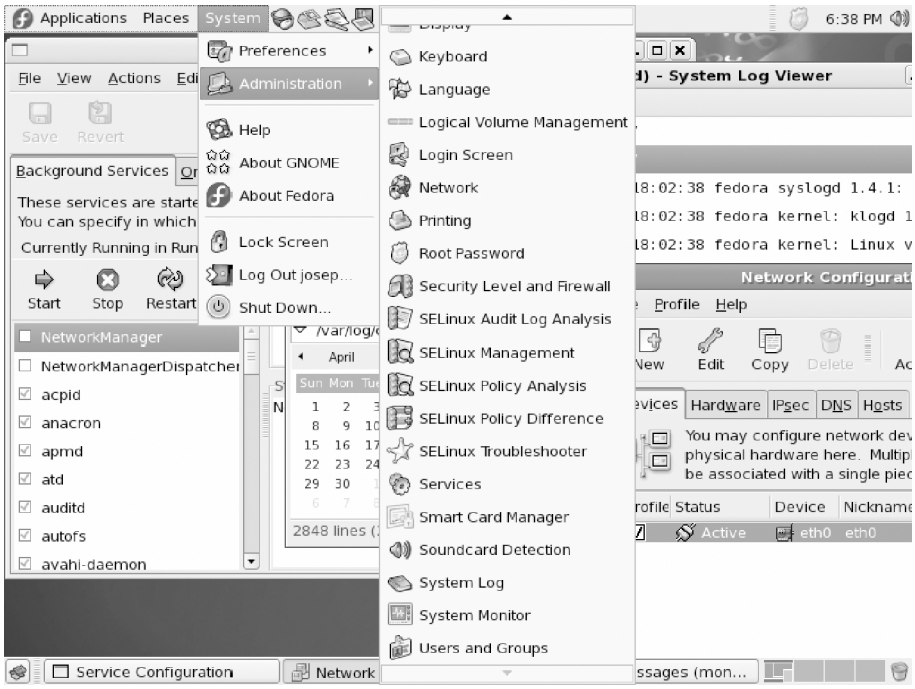

Figure 3. A few of the administration graphics utilities in Fedora

7. Other tools

In this unit's limited space we cannot comment on all the tools that can offer us benefits for administration. We will cite some of the tools that we could consider basic:

- The various basic UNIX commands: grep, awk, sed, find, diff, gzip, bzip2, cut, sort, df, du, cat, more, file, which...

- The editors, essential for any editing task, like: vi, very much used for administration tasks because of the speed of making small changes to the files. Vim is the vi compatible editor, which GNU/Linux tends to carry; it allows a syntax coloured in various languages. Emacs, a very complete editor, adapted to different programming languages (syntax and editing modes); it has a very complete environment and an X version called Xemacs. Joe, editor compatible with Wordstar. And many more...

- Scripting languages, tools for administration, like: Perl, very useful for handling regular expressions and analysing files (filtering, ordering etc.). PHP, language that is very often used in web environments. Python, another language that can make fast prototypes of applications...

- Tool for compiling and debugging high level languages: GNU `gcc` (compiler of C and C++), `gdb` (debugger), `xxgdb` (X interface for gdb), `ddd` (debugger for various languages).

Note

See material associated to the introduction course to GNU/Linux, the man pages of the commands or a tools reference such as [Stu01].

Activities

1) For a fast reading, see the FHS standard, which will help us to have a good guide for searching for files in our distribution.

2) To revise and broaden concepts and programming of shell scripts in bash, see: [Bas] [Coo].

3) For RPM packages, how would we do some of the following tasks?:

- Find out what package installed a specific command.
- Obtain a description of the package that installed a command.
- Erase a package whose full name we don't know.

Show all the files that were in the same package as a specific file.

4) Perform the same tasks as above, but for Debian packages, using APT tools.

5) Update a Debian (or Fedora) distribution.

6) Install a generic administration tool, such as Linuxconf or Webadmin for example, on our distribution. What do they offer us? Do we understand the executed tasks, and the effects they cause?

Bibliography

Other sources of reference and information

[Bas][Coo] offer a broad introduction (and advanced concepts) of programming shell scripts in bash, as well as several examples. [Qui01] discusses the different programming shells in GNU/Linux, as well as their similarities and differences.

[Deb02][Bai03] offer a broad vision of the software package systems of the Debian and Fedora/Red Hat distributions.

[Stu] is a wide introduction to the tools available in GNU/Linux.