Jose L. Muñoz
Juanjo Alins
Jorge Mata
Oscar Esparza

UPC Telematics Department

ADMINUX

Contents

Part I

Linux Essentials

Chapter 1

Introduction to Unix/Linux

1.1 Introduction to OS

This chapter provides some basic background about the Linux Operating System. In short, an Operating System (OS) is a set of software whose purpose is to **(i)** manage the resources of a computer system while **(ii)** providing an interface for the interaction with human beings.

- **Resources management**. The computer resources that can be managed by an OS include CPU, RAM, and input/output (I/O) devices. For example, in all modern computer systems, it is possible to run more than one process simultaneously. So, the OS is responsible for allocating the CPU execution cycles and RAM memory for each process. Regarding I/O devices, these include storage devices like a Hard Disk (HDD), a Compact Disk (CD), a Digital Versatile Disc (DVD), a Universal Serial Bus (USB) device, etc. but also communication devices like wired networking (Ethernet cards) or wireless networking (WIFI cards). An introduction to the organization of Linux that allows this OS to achieve a proper way of managing and accessing system resources is provided in Section 1.2.

- **User Interaction.** Another essential issue is how the OS allows interaction between the user (human being) and the computer. This interaction includes operations over the file system (copy, move, delete files), execution of programs, network configuration, and so on. The two main system interfaces for interaction between the user and the computer - CLI and GUI - are further discussed in Section 1.3.

1.2 Resources Management

1.2.1 History

Nowadays, most deployed OS come either from Microsoft WINDOWS/DOS or from UNIX. We must remark that UNIX and DOS were initially designed for different purposes. While DOS was developed for rather simple machines called Personal Computers (PC), UNIX was designed for more powerful machines called Mainframes. DOS was originally designed to run only one user process[1] or task at a time (mono-process or mono-task) and to manage only one user in the system (mono-user).

On the other hand, UNIX was directly designed as a multi-user system. This has numerous advantages, security for example, which is necessary for protection of sensitive information, was designed at the very beginning in UNIX. UNIX was designed also as multi-task being able of managing several users running several programs at the same time. As shown in Figure 1.1, today both types of OS (WINDOWS/UNIX) are capable of managing multiple users and multiple processes and also both can run over the same type of hardware. However, we would like to point out that many of these capabilities were present in the first design of UNIX, while they have been added in DOS-like systems.

[1] A process is basically a running program.

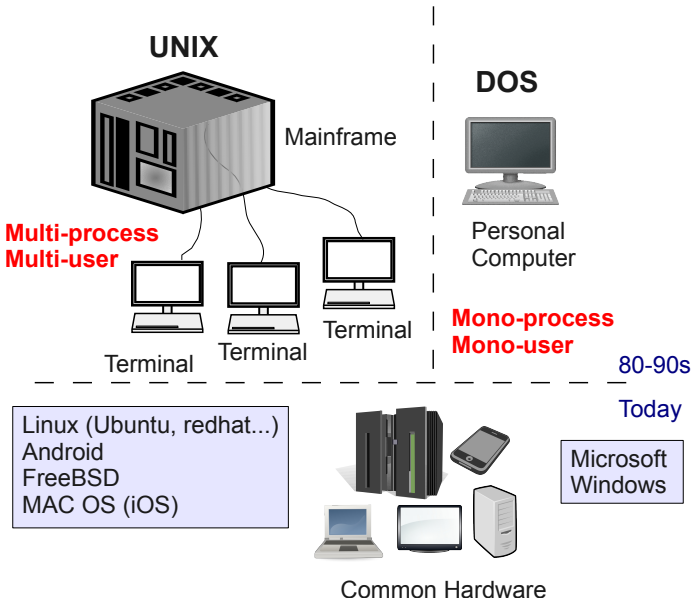


Figure 1.1: Origins of Linux.

1.2.2 OS Rings and the Kernel

Modern OS can be divided in at least three parts: **hardware**, **kernel** and **user space** (user applications or processes). See Figure 1.2.

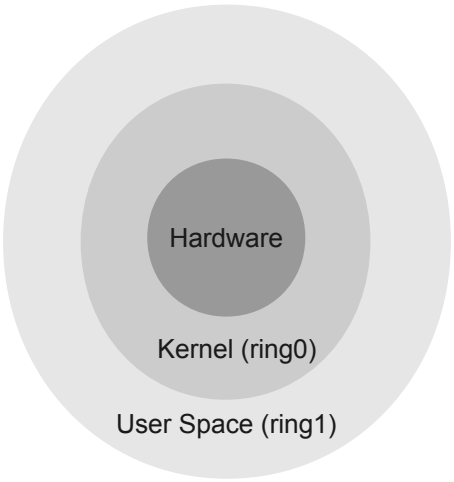


Figure 1.2: OS Rings.

These parts can be seen as “rings“. The kernel or ”ring 0” is an intermediary between user applications and the actual data processing done at the hardware level. The kernel’s primary function is to manage the computer’s resources and allow other programs to run and use these resources. These resources are at least:

- **Central Processing Unit (CPU**). The CPU is responsible executing programs. The kernel takes responsibility for deciding which of the running programs should be placed in the processor or processors.
- **Memory.** Memory is used to store both program instructions and data. For a program in execution both program

instructions and data need to be present at the memory. Often, multiple programs may want to access to memory and the kernel is responsible for deciding which memory each process can use, and determining what to do if not enough memory is available.

- **Input/Output (I/O) Devices.** These devices add some functionality to the system. Examples are keyboard, mouse, disk drives, printers, displays, etc. The kernel manages requests from user applications to perform input and output operations and provides convenient methods for using each device (typically abstracted to the point where the application does not need to know implementation details of each device).

1.2.3 System Calls

The kernel manages all the low level operations with the hardware (CPU, memory and other devices). To do so, the kernel runs in what is called "supervisor mode" which provides unrestricted access to hardware. On the other hand, the kernel provides an interface for user processes so that they can use the operating system. This interface is implemented with **system calls** (see Figure 1.3). A system call defines how a user program or user application has to request a service from the kernel. For example, a system call may allow a user program to save a file in a Hard Disk but not to write bytes at certain locations of the HDD (this operation might require supervisor mode).

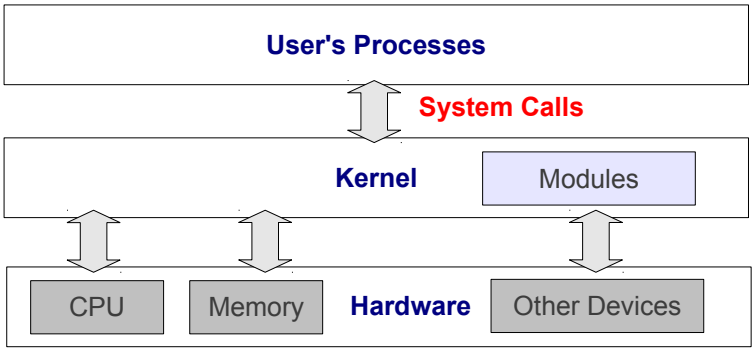


Figure 1.3: System Calls & Modular Design.

Generally, systems provide a library or API that sits between user programs and the Kernel. On Unix-like systems, that API is usually part of an implementation of the C library (libc), such as glibc, that provides wrapper functions for the system calls, often named the same as the system calls that they call. For example:

```
int kill(pid_t pid, int sig);
```

The previous function is a wrapper for a system call called kill, which is used to send a signal to a process.

1.2.4 Modules

The Linux Kernel is a **monolithic hybrid kernel**. Monolithic means that the entire operating system is working in kernel space and is alone in supervisor mode. Unlike traditional monolithic kernels, the Linux Kernel is hybrid. This means that kernel extensions, called modules, can be loaded and unloaded into the kernel upon demand while the kernel is running. In other words, modules are pieces of code that extend the functionality of the kernel without the need to reboot the system.

One type of module is the device driver, which allows the kernel to access a hardware device connected to the system. Device drivers interact with devices like hard disks, printers, network cards etc. and provide system calls.

Notice that a traditional monolithic kernel (without the possibility of having modules) has to include in its code all the possible device drivers. This has two main drawbacks: we will have larger kernels and we will need to rebuild and reboot the kernel each time we want a new functionality.

Finally, when you build a Linux kernel you can decide if you include a certain module inside the kernel (statically compiled module) or if you allow this module to be loaded at run time by your kernel (dynamic module).

The commands to list, insert and remove modules are: `lsmod`, `modprobe` and `rmmod`.

1.3 User Interaction

In the past, mainframe systems had several physical terminals connected via a serial port (often using the RS-232 serial interface). Terminals were simple monochrome displays with the minimal hardware and logic to send the text typed by the user in a keyboard and display the text received from the mainframe in the display (see Figure 1.4). On the mainframe-side, there was a command-line interpreter (CLI) or *shell*. A *shell* is a process that interprets and executes commands.

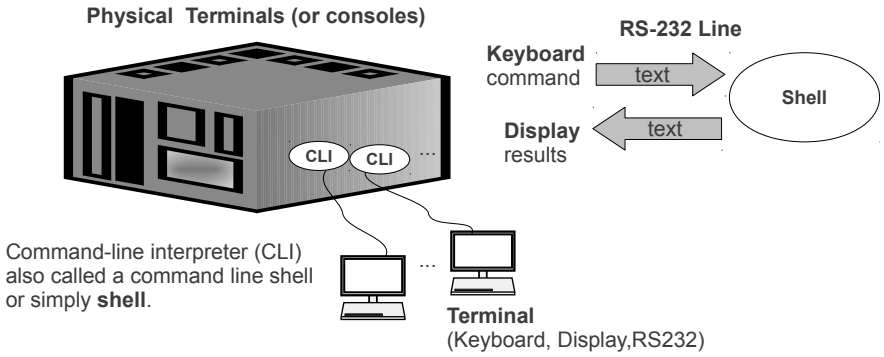


Figure 1.4: Mainframe with Old Physical Terminals.

In current systems, we have also Graphical User Interfaces or GUIs. GUI requires a graphical server (often Unix systems use the **X** server). Processes launched from the GUI are typically applications that have graphical input/output. This graphical I/O is implemented with devices such as a mouse, a keyboard, a screen, a touch screen, etc. GUIs are easier to use for novel users but in general, CLI provides you with more control and flexibility than GUI for performing system administration & configuration.

Physical terminals are not very common today, instead, **virtual consoles** are used. If you use virtual consoles to interact with our Linux system, you will not require a graphic server running on the system. Virtual consoles just manage text and they can emulate the behavior of several physical terminals. The different virtual consoles are accessed using different key combinations. By default, when Linux boots it starts six virtual consoles which can be accessed with **CRL+ALT+F1** ... **CRL+ALT+F6**. The communication between the virtual console and the *shell* is performed using a special device file in the system of the form /dev/ttyX (where X is the number of virtual console). This device file is called TTY and it emulates the "old physical communication channel". To interact with the terminal you can write to and read from the TTY. If you want to see the TTY of a terminal just type the `tty` command:

```
$ tty
/dev/tty1
```

Commands executed from a terminal are connected with a *shell* using the TTY. In the Unix jargon, it is said that the command is "attached" to a TTY.

On the other hand, we can also use a GUI if our Linux system has an graphical server running. In fact, the GUI is the default interface with the system for most desktop Linux distributions. To go from a virtual console to the graphic server, you can type **CRL+ALT+F7** in the majority of the Linux systems. Once you log into the GUI, you can also start a "terminal". In this case, the terminal is called **terminal emulator** or **pseudo-terminal**. For example, to start a pseudo-terminal you can use **ALT + F2** and then type `gnome-terminal` or `xterm`. You can also use the main menu: **MENU-> Accessories-> Terminal**. This will open a `gnome-terminal`, which is the default terminal emulator in our Linux distribution (Ubuntu). In Figure 1.5 you can observe the main features of virtual consoles and pseudo-terminals including their TTY device files. Notice that in the case of pseudo-terminals, the TTY is of the form /dev/pts/X (where X is the number of pseudo-terminal).

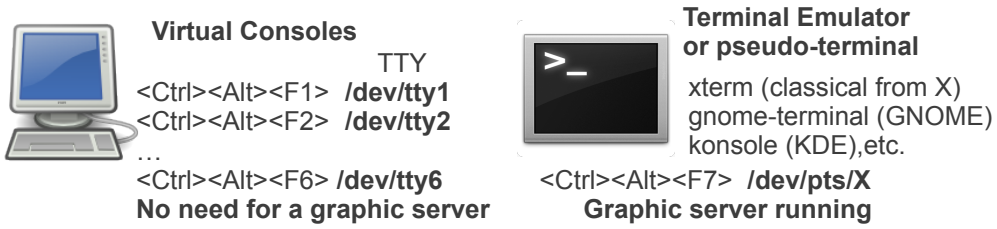


Figure 1.5: Linux Terminals.

Regarding the *shell*, this documentation refers only to Bash (Bourne Again Shell). This is because this *shell* is the most widely used one in Linux and includes a complete structured programming language and a variety of internal functions.

Note. When you open a terminal (either a virtual console or a pseudo-terminal) you will see a line of text that ends with the dollar sign "$" and a blinking cursor. When using "$" throughout this document, we will mean that we have opened a terminal with our user that is ready to receive commands.

1.4 Implementations and Distros

UNIX is now more a philosophy than a particular OS. UNIX has led to many implementations, that is to say, different UNIX-style operating systems. Some of these implementations are supported/developed by private companies like Solaris of Sun/Oracle, AIX of IBM, SCO of Unisys, IRIX of SGI or Mac OS of Apple, Android of Google, etc. Other implementations of Unix as " Linux" or "FreeBSD" are not commercial and they are supported/developed by the open source community.

In the context of Linux, we also have several distributions, which are specific forms of packaging and distributing Linux (Kernel) and its applications. These distributions include Debian, Red Hat and some derivate of these like Fedora, **Ubuntu**, Linux Mint, SuSE, etc.

1.5 Switching Users

We need some method to interact with the system as superuser (or as another user). Obviously, one possibility is to log into the system using the proper user account but it would be desirable to have commands that enable this without having to "relog". To this respect, we can use the commands `su` and `sudo`. The `su` command stands for "switch user", and allows you to become another user or execute commands as another user. For example:

```
$ su telematic
```

The previous command prompts you for the password of the user "telematic". If you don't provide a user, the `su` command defaults to the root account, which in Unix is the system administrator account. In either case, with `su` you will be prompted to enter **the password associated with the account to which you are switching**. After you execute the `su` command you will be logged as the new user until you exit. You can exit typing **`Ctrl-d`** or typing exit.

To use the `su` command on a per-command basis, you can type:

```
$ su user -c command
```

However, using `su` creates security hazards. It is potentially dangerous since it is not a good practice, for example, to have numerous people knowing and using the password of the root. Notice that when logged in as root, you can do anything in the system.

For this reason, Linux people came up with another command: `sudo`. Using the sudoers file (/etc/sudoers), system administrators can define which users or groups will be able to execute certain commands (or even any command) as root but these users will not have to know the password of the root. The `sudo` command **prompts to introduce the password of the user that is executing `sudo`** (see Figure 1.6).

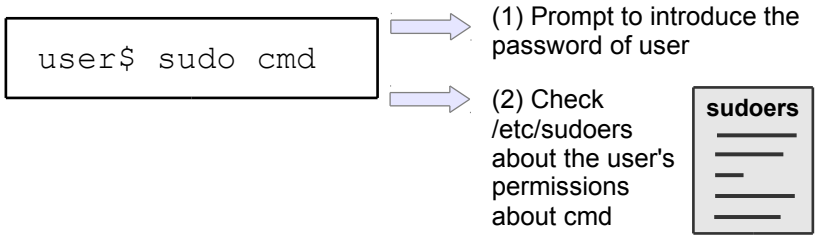


Figure 1.6: How `sudo` works.

In this way, the command `sudo` makes it easier to implement the principle of "least privilege". It also logs all commands and arguments so there is a record of who used it for what, and when. To use the `sudo` command, at the command prompt, enter:

```
$ sudo command
```

Replace command with the command for which you want to use `sudo`. If your user is configured as system administrator in the sudoers file you can get a shell as root typing:

```
user$ sudo -s
root#
```

Note. We will use $ to mean that the command is being executed as a regular user and # to mean that the command is being executed as root.

You can use the command `whoami` to know which user you are at this moment. When authenticated, a timestamp is used (stored in /var/run/sudo) and the user can use sudo without a password for 5 minutes.

1.6 Installing Software

1.6.1 Static and Dynamic Libraries

We have to understand the differences between static and dynamic libraries to fully understand the process of installing software in our Linux box (see Figure 1.7).

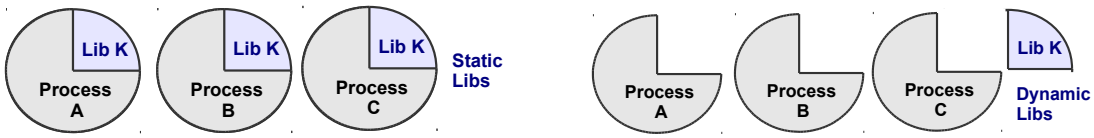


Figure 1.7: Static and dynamic libraries.

- **Static libraries**[2] or statically-linked libraries are a set of routines, external functions and variables which are resolved at compile-time and copied into the final object file by the compiler. This is called "static compiling" and program produced by this process is called a "static executable".
- With **dynamic libraries**, the kernel provides facilities for the creation and use of dynamically bound shared libraries. Dynamic binding allows external functions and variables to be referenced in user code and defined in a shared library to be resolved at run time, that is to say, when the program is loaded to become a process in the system. Therefore, the shared library code is not present in the executable image on disk. Shared code is loaded into memory once in the shared library segment and shared by all processes that reference it.

[2] In the past, libraries could only be static.

The main **advantage of static executables** is that they avoid dependency problems. Since libraries are included at compiling time, we can be sure that anything used from external libraries is available (with the correct version) before the program is executed. The main **drawback of static linking** is that the size of the executable becomes greater than in dynamic linking, as the library code is stored within the executable rather than in separate files and, which is worse, static processes in execution consume more memory than dynamically linked processes (see Figure 1.7).

On the other hand, the **main advantages of shared libraries** are that they use less disk space because the shared library code is not included in the executable programs. They also use less memory because the shared library code is only loaded once. The load time may be also reduced because the shared library code migth be already in memory. Dynamic libraries also allow the library to be updated to fix bugs and security flaws without updating the applications that use the library. The main **drawback of dynamic libraries** is that they usually establish complex relationships between the different packages of software installed in a system. For example, a configuration might enforce having several versions of the same library simultaneously installed in the system to satisfy the dependencies (required dynamic libraries) of different software packages.

For example, we are going to compile the following C program which is a typical “hello world”.

```
/* Hello World program */
#include<stdio.h>
main()
{
  printf("Hello World\n");
}
```

We can use a text editor (like `gedit`) to save the previous text as hello.c. Then, we compile this C source file to produce an executable:

```
$ gcc -o hello hello.c
$ ./hello
Hello World
```

By default the `gcc` compiler creates dynamic executables. You can check this with the `ldd` command, which shows the dependencies of a program.

```
$ ldd hello
    linux-vdso.so.1 =>  (0x00007fff5e7ca000)
    libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007fca8eca1000)
    /lib64/ld-linux-x86-64.so.2 (0x00007fca8f089000)
```

We can view the size of our executable with the `du` (disk usage) command:

```
$ du hello
12      hello
```

Compare this when we statically compile the same program:

```
$ gcc -static -o hello.static hello.c
$ ./hello.static
Hello World
$ ldd hello.static
    not a dynamic executable
$ du hello.static
860     hello.static
```

The advantages of dynamic executables are clear but the management of the different libraries on the system results in a challenge colloquially known as "dependency hell".

On Windows systems, this is called “DLL hell” (DLL comes from Dynamically Linked Library). On windows systems, it is common to distribute and install the library files that an application needs with the application itself.

On Unix-like systems, this is less common as **Package Management Systems** can be used to ensure that the correct library files are available in the system. This allows the library files to be shared between many applications leading to disk space and memory savings.

1.6.2 Sofware Packages

In Linux systems, a package of software tracks where all its files are, allowing the user to easily manage the installed software: view dependencies, uninstall, etc. Generally, in Linux, software packages copy their executables in /usr/bin, their libraries in /usr/lib and their documentation in /usr/share/doc/package/. It's worth noting here that to take full advantage of packages, one should not go installing or deleting files behind the package system's back.

There are multiple different package systems in the Linux world, the two main ones being:

- Red Hat Packages (**.rpm** files).
- Debian Packages (**.deb** files).

Packages can be managed with the commands `rpm` and `dpkg` for rpm and deb packages, respectively.

1.6.3 Advanced Package Management Systems

The package systems previously described do not manage dependencies. So, if you need to install a package that has dependencies, you have to manually install the proper versions of the packages containing the dynamic libraries on which your package depends on (at least all the libraries not currently installed on your system).

In this context, a new generation of package management systems was developed to make this tedious process easier for the user. These advanced package management systems automatically manage package dependencies. For rpm files we have `yum` and for .deb files we have `apt`. With these tools one can essentially say "install this package" and all dependent packages will be installed/upgraded as appropriate.

Of course, one has to configure where these tools must go to find out the software packages. These packages are online in **package repositories**. Thus, you have to configure the addresses of the repositories you are interested in.

In the case of Debian packages, APT uses files that lists the 'sources' from which packages can be obtained. These files are in the directory /etc/apt. APT requires to execute `apt-get update` to update the available list of packages available in online repositories. If this command is not executed, apt works with the local cache, which might be outdated.

On the other hand, in an Ubuntu system, we can type the name of an application in the console and, if it is not installed, the system will tell us how to install it:

```
$ xcowsay
The program 'xcowsay' is currently not installed.  You can install it by typing:
sudo apt-get install xcowsay
```

Then, to install the latest version of `xcowsay`, you can update first and then install:

```
$ sudo apt-get update
$ sudo apt-get install xcowsay
```

You can download (only) the required packages to install an application (in this example `gparted`):

```
$ sudo apt-get download gparted libparted-fs-resize0
$ sudo mv *.deb /var/cache/apt/archives
```

Then, when you want to install `gparted`:

```
$ sudo apt install gparted
```

The previous command does not need an online connection.

You can remove a package with:

```
$ sudo apt-get remove gparted
```

You did remove the software but typically, there are files associated with it such as configuration files and folders. To remove them:

```
$ sudo apt-get remove gparted --purge
```

1.6.4 Installing from the Source

When you need to install software that is neither in a repository or has an individual package created, you can install from the source code. This is compatible with all Linux distributions. The source code typically contains a bunch of files of the application, packed in a .tar archive and compressed using GNU Zip (.gz) or BZip2 (.bz2). Format: `<filename>.tar.gz or <filename>.tar.bz2` These types of files can be unzipped and unpacked on a directory using the `tar` command:

```
$ tar xvzf <filename>.tar.gz
$ tar xvjf <filename>.tar.bz2
```

By convention, there are files called "INSTALL" or "README" giving application-specific usage information. The typical compilation/installation steps are:

1. Unpack the tar archive (tarball):

   ```
   $ tar xzvf <package_name>.tar.gz
   $ tar xvjf <package_name>.tar.bz2
   ```

2. Change to the extracted directory

   ```
   $ cd <extracted_dir_name>
   ```

3. Run source configuration script as follows:

   ```
   $ ./configure
   ```

4. Build the source code using the GNU Make utility. In the directory of the sources there will be a Makefile file describing how to compile. You can call make (compile) as follows:

   ```
   $ make
   ```

5. Install the package as follows:

   ```
   # make install
   ```

As a final remark, we would like to mention that in general, you should avoid installing software from the source if there is a package available[3].

1.6.5 Command Summary

The table 1.1 summarizes the commands used within this section.

[3] Many times it is not too hard to make the package yourself.

Table 1.1: Summary of commands.

`lsmod`	list current system modules.
`insmod` or `modprobe`	insert a module.
`rmmod`	remove a module.
`tty`	view the associated TTY file of the terminal.
`gnome-terminal` or `xterm`	terminals.
`su`	switch to another user.
`sudo`	execute something as another user using sudoers file.
`gcc`	c compiler .
`ldd`	list dynamic libraries dependencies.
`du`	check the disk usage of files.
`dpkg`	debian packages manager.
`apt-get`	advanced package manager for debian packages.
`apt-file`	search the package that installed a certain file.

Chapter 2

Processes

2.1 Booting the System

A sequence of steps is followed to boot (start) a Linux system in which the control first goes to the BIOS, then to a boot loader and finally, to a Linux kernel (the system core). The boot loader is not absolutely necessary. Certain BIOS can load and pass control to the Linux kernel without the use of the loader but in practice this element is usually always present. Figure 2.1 illustrates the boot process.

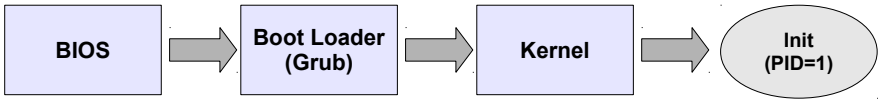


Figure 2.1: The Linux Boot Process.

Once the kernel is started, it executes the first process: *init*. A process is the abstraction used by the operating system to represent a running program. Each process in Linux consists of an address space and a set of data structures within the kernel. The address space contains the code and libraries that the process is executing, the process's variables, its stacks, and different additional information needed by the kernel while the process is running. The kernel also implements a "scheduler" to share the CPU resources available in the system among the different processes.

Linux processes have "kinship". The process that generates another process is called **parent** process. The process generated by another process is called **child** process. The processes can be parent and child both at the same time. Therefore, you can see the processes on a Linux system hierarchically organized in a tree. The root of the "tree of processes" is *init*. Finally, each process has a unique identifier called "Process ID" or PID. The PID of *init* is 1.

2.2 Listing Processes

The command `ps` provides information about the processes running on the system. If we open a terminal and type `ps`, we obtain a list of running processes. In particular, those launched from the terminal.

```
$ ps
  PID TTY          TIME CMD
21380 pts/3    00:00:00 bash
21426 pts/3    00:00:00 ps
```

Let's see what these columns mean:

- The first column is the process identifier.
- The second column is the associated terminal[1]. In the previous example is the pseudo-terminal 3. A question mark (?) in this column means that the process is not associated with a terminal.

[1]The terminal can also be viewed with the `tty` command.

- The third column shows the total amount of time that the process has been running.
- The fourth column is the name of the process.

In the example above, we see that two processes have been launched from the terminal: the `bash`, which is the shell, and the command `ps` itself. Figure 2.2 shows a scheme with the relationships of all the processes involved when we type a command in a terminal.

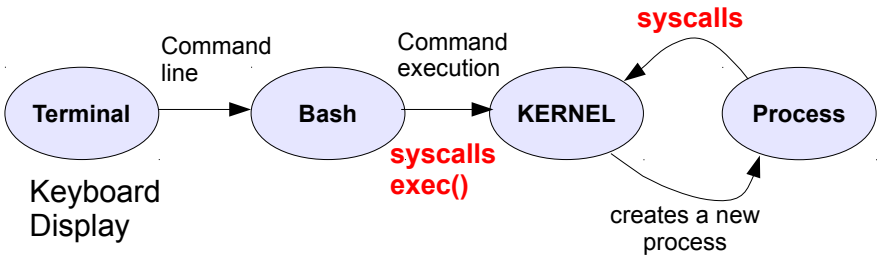


Figure 2.2: Processes related with a pseudo-terminal.

On the other hand, the command `ps` accepts parameters. For example the parameter -u reports all the processes launched by a user. Therefore, if you type:

```
$ ps -u user1
```

You will obtain a list of all the processes owned by `user1`. Some relevant parameters of `ps` are:

- ***-A*** shows all the processes from all the users.
- ***-u user*** shows processes of a particular user.
- ***-f*** shows extended information.
- ***-o format*** the format is a list separated by commas of fields to be displayed.

Examples:

```
$ ps -Ao  pid,ppid,state,tname,%cpu,%mem,time,cmd
$ ps -u user1 -o pid,ppid,cmd
```

The preceding command shows the process PID, the PID of parent process (PPID), the state of the process, the associated terminal, the % of CPU and memory consumed by the process, the accumulated CPU time consumed and the command that was used to launch the process.

2.3 The `man` Command

The `man` command shows the "manual" of other commands. Example:

```
$ man ps
```

If you type this, you will get the manual for the command "`ps`". Once in the help environment, you can use the arrow keys or `AvPag`/`RePaq` to go up and down. To search for text *xxx*, you can type `/xxx`. Then, to go to the next and previous matches you can press keys `n` and `p` respectively. Finally, you can use `q` to exit the manual.

2.4 Working with the Terminal

Bash keeps a history the commands typed. The history can be seen with the command `history`. You can retype a command of the history using `!number`. You can also press the up/down arrow to scroll back and forth through your command history.

On the other hand, the bash provides another useful feature called command line completion (also called tab completion). This is a common feature of bash and other command line interpreters. When pressing the `TAB`, bash automatically fills in partially typed commands or parameters.

Finally, X servers provide a "copy and paste" mechanism. You can easily copy and paste between pseudo-terminals and applications using your mouse. Most Linux distributions are configured in such a way that a click with the mouse's middle button (or scroll wheel) is interpreted as a paste operation of the selected text. If your mouse has only two buttons, hit both buttons simultaneously to emulate the middle button. Typically, the combinations `CRL+SHIFT+c` and `CRL+SHIFT+v` also work for copy and paste.

2.5 Other Commands for Processes

The **`pstree`** command displays all the system processes within a tree showing the relationships between processes. The root of the tree is either init or the process with the given PID.

The **`top`** command returns a list of processes in a similar way as `ps` does, except that the information displayed is updated periodically so we can see the evolution of a process' state. **`top`** also shows additional information such as memory space occupied by the processes, the memory space occupied by the exchange partition or swap, the total number of tasks or processes currently running, the number of users, the percentage processor usage, etc.

Finally, the **`time`** command gives us the duration of execution of a particular command. Example:

```
$ time ps
  PID TTY          TIME CMD
 7333 pts/0    00:00:00 bash
 8037 pts/0    00:00:00 ps

real    0m0.025s
user    0m0.016s
sys     0m0.012s
```

Real refers to actual elapsed time; User and Sys refer to CPU time used only by the process.

- Real is wall clock time - time from start to finish of the call. This is all elapsed time including time slices used by other processes and time the process spends blocked (for example if it is waiting for I/O to complete).
- User is the amount of CPU time spent in user-mode code (outside the kernel) within the process. This is only actual CPU time used in executing the process. Other processes and time the process spends blocked do not count towards this figure.
- Sys is the amount of CPU time spent in the kernel within the process. This means executing CPU time spent in system calls within the kernel, as opposed to library code, which is still running in user-space. Like 'user', this is only CPU time used by the process.

Notice that User+Sys will tell you how much actual CPU time your process used. This is across all CPUs, so if the process has multiple threads it could potentially exceed the wall clock time reported by Real.

2.6 Scripts

In our system we will have programs. The source of a program is first compiled, and the result of that compilation is executed to become a process in the system. The process directly interacts with the system kernel. Examples of languages to build programs: C, C++, etc. On the other hand, a script is interpreted. In other words, a script is written to be understood by an interpreter (see Figure 2.3). Scripting examples are shell scripts, Python scripts, PHP scripts, scripts for Javascript, etc. Typically scripts are written for small applications and they are easier to develop. However, scripts are also usually slower than programs due to the interpretation process. In this case, the interpreter is who finally interacts with the system kernel.

In particular, a shell script is a text file containing commands and special internal shell commands (if,for, while, etc.) that have to be interpreted. As its mane suggests, a shell (bash in most Linux systems) is who interprets and executes a shell script.

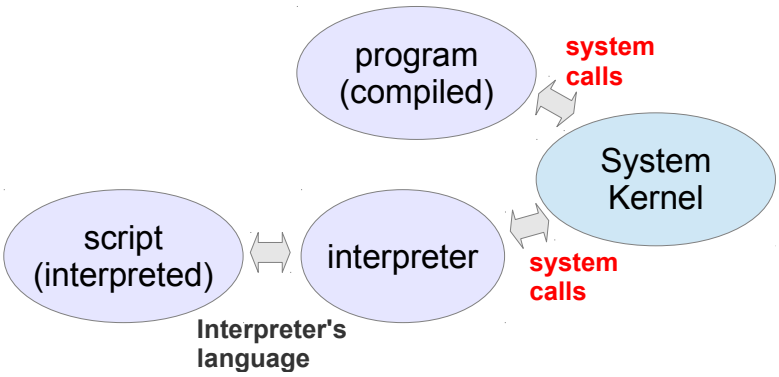


Figure 2.3: Program versus Script.

Why to write shell scripts? They are useful to create your own commands, they save your time, some tasks of your daily life can be automated and system administration can be automated too.

You can use any text editor like `vi` (CLI) or `gedit` (GUI) to write down a script. Then, you must give it execution permissions to be able to execute it. A simple example is:

```
ps
sleep 2
pstree
```

Then, we give execution permissions to our user and execute it:

```
$ chmod u+x firstscript.sh
$ ./firstscript.sh
  PID TTY          TIME CMD
 5936 pts/2    00:00:00 bash
 5996 pts/2    00:00:00 bash
 5997 pts/2    00:00:00 ps
init -- /usr/bin/x-term -- /usr/bin/x-term
     |                   | --bash--bash--pstree
    ...
```

The previous script command executes a `ps` and then, after approximately 2 seconds (sleep makes us wait 2 seconds) a `pstree` command. You can notice from the output of the previous script that the `bash` clones itself to execute the commands within the script. We can say that the "script PID" is the PID of the cloned (child) `bash`. This is a way of protecting the parent bash in case anything in the script goes wrong. As we will see next, we can kill processes using their PID. Another example script is the classical "Hello world" script.

```
#!/bin/bash
# Our second script, Hello world!
echo Hello world
```

As you can observe, the script begins with a line that starts with "#!". This is the path to the shell that will execute the script. In general, you can ignore this line (as we did in our previous example) and then the script is executed by the default shell. However, it is considered a good practice to always include it in scripts. In our case, we will build scripts always for the `bash` shell.

As you can observe, we can use the `echo` command to write to the terminal. Finally, you can also observe that text lines (except the first one) after the sign "#" are interpreted as comments.

Next, we assign execution permission and execute the script:

```
$ chmod u+x secondscript.sh
$./secondscript.sh
Hello world
```

Finally, in a script you can also read text typed by the user in the terminal. To do so, you can use the `read` command. For example, try:

```
#!/bin/bash
# Our third script, using read for fun
echo Please, type a sentence and hit ENTER
read TEXT
echo You typed: $TEXT
```

In this script we can observe some new things. The text introduced by the user in the terminal is stored in a variable called "TEXT" by the `read` command. When we want to use the content (value) of a variable, we must set the sign "$" before the name of the variable. Notice that we use the value of the variable TEXT in the `echo` command in this script.

2.7 Running Processes in Foreground/Background

By default, the `bash` executes commands interactively, i.e., the user enters a command, the `bash` marks it to be executed, waits for the output and once concluded returns the control to the user, so a new command can be executed. This type of execution is also called a **foreground command execution**. For example:

```
$ xeyes
```

This application prints eyes that follow the mouse movements. While running a command in foreground we cannot run any other commands. However, `bash` also let us run commands non-interactively or in **background**. To do so, we must use the ampersand symbol (&) at the end of the command line. Example:

```
$ xeyes &
```

Whether a process is running in foreground or background, its output goes to its attached terminal. However, a process cannot use the input of its attached terminal while running in background.

Note. We can try this with our thirdscript.sh.

2.8 Signals

A signal is a limited form of inter-process communication using the system kernel and the `kill()` syscall. Actually, a signal is just an integer. As you can observe in Figure 2.4, some signals are destined for the kernel (non-capturable signals) and other signals are destined for user-space processes (capturable signals).

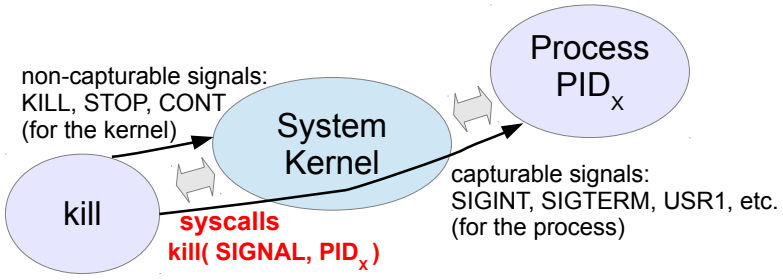


Figure 2.4: `Kill` command and `kill()` syscall.

Capturable signals are used as an asynchronous notification: when a signal is received by a user-space process, it interrupts its normal flow of execution and executes the corresponding "signal handler" (function) based on the signal number. The `kill` command can be used to send signals [2].

[2] More technically, the command `kill` is a *wrapper* around the system call `kill()`, which can send signals to processes or groups of processes in the system, referenced by their process IDs (PIDs) or process group IDs (PGIDs).

The default signal that the `kill` command sends is the termination signal (`SIGTERM`), which asks the process for releasing its resources and exit. The integer and name of signals may change between different implementations of Unix. Usually, the `SIGKILL` signal is the number 9 and `SIGTERM` is 15. You can view the signals defined in your system with `kill -l`. In Linux, the most widely used signals and their corresponding integers are:

- 9 `SIGKILL`. Non-capturable signal sent to the kernel to end a process immediately.
- 20 `SIGSTOP`. Non-capturable signal sent to the kernel to stop a process. This signal can be generated in a terminal for a process in foreground pressing **`Control-Z`**.
- 18 `SIGCONT`. Non-capturable signal sent to the kernel that resumes a previously stopped process. This signal can be generated typing `bg` in a terminal.
- 2 `SIGINT`. Capturable signal sent to a process to tell it that it must terminate its execution. It is sent in an interactive terminal for the process in foreground when the user presses **`Control-C`**.
- 15 `SIGTERM`. Capturable signal sent to a process to ask for termination. It is sent from the GUI and also this is the default signal sent by the `kill` command.
- `USR1`. Capturable signal that can be used for any desired purpose.

The `kill` command syntax is: **`kill -signal PID`**. You can use both the number and the name of the signal:

```
$ kill -9 30497
$ kill -SIGKILL 30497
```

In general, signals can be intercepted by processes, that is, processes can provide a special treatment for the received signal. However, SIGKILL, SIGSTOP and SIGCONT cannot be captured. They are destined to the kernel, which kills, stops or resumes the process. This provides a safe mechanism to control the execution of processes. SIGKILL ends the process and SIGSTOP pauses it until a SIGCONT is received.

Unix has also security mechanisms to prevent an unauthorized users from finalizing other user processes. Basically, a process cannot send a signal to another process if both processes do not belong to the same user. Obviously, the exception is the user *root* (superuser), who can send signals to any process in the system.

Finally, another interesting command is **`killall`**. This command is used to terminate execution of processes by name. This is useful when you have multiple instances of a running program.

2.9 Job Control

Job control refers to the bash feature of managing processes as jobs. For this purpose, bash provides the commands **`"jobs"`**, **`"fg"`**, **`"bg"`** and the hot keys **`Control-z`** and **`Control-c`**.

The command `jobs` displays a list of processes launched from a specific instance of bash. Each instance of bash considers that it has launched processes as *jobs*. Each job is assigned an identifier called a JID (Job Identifier).

Let's see how the job control works using some examples and a couple of graphic applications: `xeyes` and `xclock`. The application `xeyes` shows eyes that follow the mouse movement, while `xclock` displays a clock in the screen.

```
$ xeyes &
$ xeyes &
$ xclock &
$
```

The previous commands run 2 processes or instances of the application `xeyes` and an instance of `xclock`. To see their JIDs, you can type `jobs`.

```
$ jobs
[1]   Running                 xeyes &
[2]-  Running                 xeyes &
[3]+  Running                 xclock &
$
```

In this case we can observe that the JIDs are 1, 2 and 3 respectively. Using the JID, we can make a job to run in *foreground*. The command is `fg`. The following command brings to *foreground* job 2.

```
$ fg 2
xeyes
```

On the other hand, the combination of keys **Control-z** sends a stop signal (SIGSTOP) to the process that is running on *foreground*. Following the example above, we had a *job* `xeyes` in the foreground, so if you type **`Control-z`** the process will be stopped.

```
$ fg 2
xeyes
^Z
[2]+  Stopped                 xeyes
$ jobs
[1]   Running                 xeyes &
[2]-  Running                 xclock &
[3]+  Stopped                 xeyes
$
```

To resume the process that we just stopped, type the command `bg`:

```
$ bg
[2]+ xeyes &
$
```

Using the JID after the command `bg` will send the corresponding job to *background* (in the previous case we can use `bg 2` to produce the same result). The JID can also be used with the command `kill`. To do this, we must write a % sign right before the JID to differentiate it from a PID. For example, we could terminate the job "1" using the command:

```
 $ kill -s SIGTERM %1
```

Another very common shortcut is **Control-c** and it is used to send a signal (SIGINT) to terminate the process that is running on *foreground*. Example:

```
$ fg 3
xclock
^C
[1]   Terminated              xeyes
```

Notice that whenever a new process is run in *background*, the bash provides us the JID and the PID:

```
$ xeyes &
[1] 25647
```

Here, the job has JID=1 and PID=25647.

2.10 Capturing Signals in Scripts: `trap`

The `trap` command allows the user to catch signals in bash scripts. If we use this script:

```
trap "echo I do not want to finish!!!!" SIGINT
while true
do
sleep 1
done
```

Try to press **`Control-c`**.

2.11 Running Multiple Commands

If a command is successfully executed it returns a "0". If an error occurred, the command must return a positive value that provides some information about the error (many times it is just a "1"). The "return code" of the previously executed command is stored in the Bash in a special variable called "?". To view the content of this variable you can type `echo $?`.

```
$ ps -k
$ echo $?
$ ps
$ echo $?
```

Using return codes, we can also use different ways of executing commands:

- **`$ command`** the command runs in the foreground.
- **`$ command1 & command2 & ... commandN &`** commands will run in background.
- **`$ command1; command2; ... ; commandN`** sequential execution of commands.
- **`$ command1 && command2 && ... && commandN`** `commandX` is executed if the last executed command has exit successfully (return code 0).
- **`$ command1 || command2 || ... || commandN`** `commandX` is executed if the last executed command has NOT exit successfully (return code >0).

2.12 Extra

2.12.1 *Threads

Threads are a popular programming abstraction for parallel execution on modern operating systems. When threads are forked inside a program for multiple flows of execution, these threads share certain resources (e.g., memory address space, open files) among themselves to minimize forking overhead and avoid expensive inter-process communication (IPC) channel. These properties make threads an efficient mechanism for concurrent execution.

In Linux, threads, also called Lightweight Processes (LWP), created within a program, will have the same "thread group ID" as the program's PID. Each thread will then have its own thread ID (TID). To the Linux kernel's scheduler, threads are nothing more than standard processes which happen to share certain resources. Classic command-line tools such as `ps` or `top`, which display process-level information by default, can be instructed to display thread-level information. Example:

```
$ ps -T -p 2741
 PID  SPID TTY          TIME CMD
2741  2741 ?        00:00:01 nm-applet
2741  2754 ?        00:00:00 dconf worker
2741  2760 ?        00:00:00 gdbus
```

2.12.2 *Terminal Associated Signals

A shell process is a child of a terminal and when we execute a command, the command becomes a child of the shell. If the terminal is killed or terminated (without typing exit), a SIGHUP signal (hang up) sent to all the processes using the terminal (i.e. bash and currently running commands).

```
$ tty
/dev/pts/1
$ echo $PPID
11587
$ xeyes &
```

```
[1] 11646
$ ps
  PID TTY          TIME CMD
11592 pts/1    00:00:00 bash
11646 pts/1    00:00:02 xeyes
11706 pts/1    00:00:00 ps
```

If you close the terminal from the GUI, or if you type the following from another terminal:

```
$ kill -TERM 11587
```

You will observe that the `xeyes` process also dies. However, if you type exit in a shell with a process in background, you will notice that the process does not die but it becomes *orphan* and `Init` is assigned as its new parent process. There is a shell utility called `nohup`, which can be used as a wrapper to start a program and make it immune to SIGHUP. Also the output is stored in a file called nohup.out in the directory from which we invoked the command or application. Try to run the previous example with `$ nohup xeyes &`.

On the other hand, if there is an attempt to read from a process that is in background, the process will receive a SIGTTIN signal. If not captured, this signal suspends the process. Finally, if you try to exit a bash while there are stopped jobs, it will alert us. Then the command **`jobs`** can be used to inspect the state of all those jobs. If `exit` is typed again, the warning message is no longer shown and all suspended tasks are terminated.

2.12.3 *States of a Process

A process might be in several states:

- Ready (R) - A process is ready to run. Just waiting for receiving CPU cycles.
- Running (O) - Only one of the ready processes may be running at a time (for uniprocessor machine).
- Suspended (S) - A process is suspended if it is waiting for something to happen, for instance, if it is waiting for a signal from hardware. When the reason for suspension goes away, the process becomes Ready.
- Stopped (T) - A stopped process is also outside the allocation of CPU, but not because it is suspended waiting for some event.
- Zombie (Z) - A zombie process or *defunct* is a process that has completed execution but still has an entry in the process table. Entries in the process table allow a parent to end its children correctly. Usually the parent receives a SIGCHLD signal indicating that one of its children has died. Zombies running for too long may point out a problem in the parent source code, since the parent is not correctly finishing its children.

Note. A zombie process is not like an “orphan” process. An orphan process is a process that has lost its father during its execution. When processes are “ orphaned”, they are adopted by “ *Init*.”

2.12.4 *Priorities: `nice`

Each Unix process has a priority level ranging from -20 (highest priority) to 19 (lowest priority). A low priority means that the process will run more slowly or that the process will receive less CPU cycles from the kernel scheduler.

The `top` command can be used to easily change the priority of running processes. To do this, press ”r“ and enter the PID of the process that you want change it’s priority. Then, type the new level of priority. We must take into consideration that only the superuser ”root“ can assign negative priority values.

You can also use `nice` and `renice` instead of `top`. Examples.

```
$ nice -n 2 xeyes &
[1] 30497
$ nice -n -2 xeyes
nice: cannot set niceness: Permission denied
$ renice -n 8 30497
30497: old priority 2, new priority 8
```

2.12.5 Command Summary

The table 2.1 summarizes the commands used within this section.

Table 2.1: Summary of commands for process management.

man	is the system manual page.
ps	displays information about a selection of the active processes.
tty	view the associated terminal.
pstree	shows running processes as a tree.
top	provides a dynamic real-time view of running processes.
time	provides us with the duration of execution of a particular command.
sleep	do not participate in CPU scheduling for a certain amount of time.
echo	write to the terminal.
read	read from the terminal.
jobs	command to see the jobs launched from a shell.
bg and fg	command to set a process in background or foreground.
kill	command to send signals.
killall	command to send signals by name.
nice and renice	command to adjust niceness, which modifies CPU scheduling.
trap	process a signal.
nohup	command to create a process which is independent from the father process.

2.13 Practices

Exercise 2.1– In this exercise you will practice with process execution and signals.

1. Open a pseudo-terminal and execute the command to see the manual of `ps`. Once in the manual of the `ps` command, search and count the number of times that appears the pattern *ppid*.

2. Within the same pseudo-terminal, execute `ps` with the appropriate parameters in order to show the PID, the *tty* and the command of the currently active processes that have been executed from the terminal. Do the same in the virtual console number two (/dev/tty2).

3. Execute the following commands:

```
$ ps -o pid,comm
$ ps -o pid,cmd
```

   Comment the differences between the options: *cmd* and *comm*.

4. Use the `pstree` command to see the process tree of the system. Which process is the father of `pstree`? and its grandfather? and who are the rest of its relatives?

5. Open a gnome-terminal and then open a new "TAB" typing CRL+SHIFT+t. Now open another gnome-terminal in a new window. Using `pstree`, you have to comment the relationships between the processes related to the terminals that we opened.

6. Type ALT+F2 and then type `xterm`. Notice that this sequence opens another type of terminal. Repeat the same sequence to open a new `xterm`. Now, view the process tree and comment the differences with respect to the results of the previous case of gnome terminals.

7. Open three gnome-terminals. These will be noted as t1, t2 and t3. Then, type the following:

```
t1$ xeyes -geometry 200x200 -center red
t2$ xclock &
```

   Comment what you see and also which is the type of execution (foreground/background) on each terminal.

8. For each process of the previous applications (`xeyes` and `xclock`), try to find out the PID, the execution state, the *tty* and the parent PID (PPID). To do so, use the third terminal (t3).

9. Using the third terminal (t3), send a signal to terminate the process xeyes.

10. Type `exit` in the terminal t2. After that, find out who is the parent process of `xclock`.

11. Now send a signal to kill the process `xclock` using its PID.

12. Execute an `xclock` in *foreground* in the first terminal t1.

13. Send a signal from the third terminal to stop the process `xclock` and then send another signal to let this process to continue executing. Is the process executing in *foreground* or in *background*? Finally, send a signal to terminate the `xclock` process.

14. Using the job control, repeat the same steps as before, that is, executing `xclock` in *foreground* and then stopping, resuming and killing. List the commands and the key combinations you have used.

15. Execute the following commands in a pseudo-terminal:

```
$ xclock &
$ xclock &
$ xeyes &
$ xeyes &
```

    Using the job control set the first `xclock` in *foreground*. Then place it back in *background*. Kill by name the two `xclock` processes and then the `xeyes` processes. List the commands and the key combinations you have used.

16. Create a command line using execution of multiple commands that shows the processes launched from the terminal, then waits for 3 seconds and finally shows again the processes launched from the terminal.

17. Using multiple commands execution (&&, ||, etc.) create a command line that executes a `ps` command with an unsuccessful exit state and then as a result another `ps` command without parameters is executed.

18. Discuss the results of the following multiple command executions:

```
$ sleep || sleep || ls
$ sleep && sleep --help || ls && ps
$ sleep && sleep --help || ls || ps
```

Exercise 2.2– (*) This exercise deals with additional aspects about processes.

1. Create a script that asks for a number and displays the number multiplied by 7. Note. If you use the variable VAR to read, you can use `$[VAR * 7]` to display its multiplication.

2. Add signal managment to the previous script so that when the USR1 signal is received, the script prints the sentence "waiting operand".

    Tips: use `trap` **to capture USR1 and** `kill -USR1 PID` **to send this signal.**

3. Type a command to execute an `xeyes` application in background with "niceness" (priority) equal to 18. Then, type a command to view the command, the PID and the priority of the `xeyes` process that you have just executed.

Chapter 3

Filesystem

3.1 Introduction

File Systems (FS) define how information is stored in data units like HDD, DVDs, USB devices, tapes, etc. The base of a FS is the file. There's a saying in the Linux world that "everything is a file" (a comment attributed to Ken Thompson, one of the developers of UNIX). That includes directories.

Directories are just files that contain a list of other files. Files and directories are organized into a hierarchical file system, starting from the root directory / and branching out. On the other hand, files must have a name, which is tight to the following rules:

- Must be between 1 and 255 characters
- All characters can be used except for the slash "/".
- The following characters can be used in names of files but they are not recommended because they have special meanings:
  `= \ ^ ~ ' " ` * ; - ? [ ] ( ) ! & ~ < >`
- The file names are case sentive, that is to say, characters in uppercase and lowercase are distinct. For example: `letter.txt`, `Letter.txt` or `letter.Txt` do not represent the same files.

The simplest example of file is that used to store data like text, images, etc. The different FS technologies are implemented in the kernel either statically or as modules. FS define how the kernel manages files including aspects such as which meta-data is used for files, when and how a file is read or written, etc.

Examples of Disk File Systems (DFS) are reiserFS, ext2, ext3 and ext4. These are developed within the Unix environment for HDD and USB devices. For DVDs we have a file system called UDF (universal disk format). In Windows environments we have other DFS like fat16, fat32 and ntfs.

3.2 Basic types of files

Unix uses the abstraction of "file" for many purposes and thus, as mentioned, this is a fundamental concept in Unix systems. This type of abstraction allows using the API of files for devices like for example a printer. In this case, the API of files is used for writing into the file that represents the printer, which indeed means printing. Unix kernels manage three basic types of files:

- **Regular files**. These files contain data.
- **Directory files (folders)**. These files contain a list of other files. Directories are used to group other files in an structured manner.
- **Special Files**. Within this category there are several sorts of files which have some special content used by the OS.

The command `stat` can be used to view the basic type and some metadata of a file.

```
$ stat /etc/services
  File: `/etc/services'
  Size: 19281           Blocks: 40         IO Block: 4096   regular file
Device: 801h/2049d      Inode: 3932364     Links: 1
Access: (0644/-rw-r--r--)  Uid: (    0/    root)   Gid: (    0/    root)
Access: 2012-09-24 22:06:53.249357692 +0200
Modify: 2012-02-13 19:33:04.000000000 +0100
Change: 2012-05-11 12:03:24.782168392 +0200
 Birth: -
```

Everything is a file in Unix sytems. For example, the TTY file is a special file:

```
$ tty
/dev/pts/2
$ stat /dev/pts/2
  File: `/dev/pts/2'
  Size: 0               Blocks: 0          IO Block: 1024   character special file
Device: bh/11d  Inode: 5           Links: 1     Device type: 88,2
Access: (0620/crw--w----)  Uid: ( 1000/ jlmunoz)   Gid: (    5/     tty)
Access: 2014-03-25 18:11:04.152159417 +0100
Modify: 2014-03-25 18:11:04.152159417 +0100
Change: 2014-03-25 18:10:54.152159417 +0100
 Birth: -
```

3.3 Hierarchical File Systems

The "Linux File Tree" follows the FHS (Filesystem Hierarchy Standard). This standard defines the main directories and their content for GNU/Linux OS and other Unix-alike OS. In contrast to Windows variants, the Linux File Tree is not tied up to the hardware structure. Linux does not depend on the number of hard disks the system has (c:\, d:\ or m:\...). The whole Unix file system has a unique origin: the root (*/*). Below this directory we can find all files that the OS can access to. Some of the most significant directories in Linux (FHS) are detailed next:

/ File system root.

/dev Contains system files which represent devices physically connected to the computer.

/etc Contains system configuration files.
This directory cannot contain any binary files (such as programs or commands).

/lib Contains necessary libraries to run programs or commands.

/proc Contains special files which receive or send information to the kernel.
If necessary, it is recommended to modify these files with "special caution".

/bin Contains binaries of common system commands.

/sbin Contains binaries of administration commands which can only be executed by the system admin or superuser (*root*).

/usr This directory contains the common programs that can be used by all the system users.
The structure is the following:

- /usr/bin General purpose programs (including C/C++ compiler).
- /usr/doc System documentation.
- /usr/etc Configuration files of user programs.
- /usr/include C/C++ heading files (.h).
- /usr/info GNU information files.
- /usr/lib Libraries of user programs.
- /usr/man Manuals to be accessed by command *man*.
- /usr/sbin System administration programs.
- /usr/src Source code of those programs.

Additionally other directories may appear within */usr*, such as directories of installed programs.

/var Contains temporal data of programs (this doesn't mean that the contents of this directory can be erased).

/mnt or /media Contains mounted systems of pendrives or external disks.

/home Contains the working directories of the users of the system except for root.

3.4 Storage Devices

In UNIX systems, the kernel automatically detects and maps storage devices in the /dev directory. The name that identifies a storage device follows the following rules:

1. If there is an IDE controller:
   - `hda` to IDE bus/connector 0 master device
   - `hdb` to IDE bus/connector 0 slave device
   - `hdc` to IDE bus/connector 1 master device
   - `hdd` to IDE bus/connector 1 slave device

   For example, if a CD-ROM or DVD is plugged to IDE bus/connector 1 master device, Linux will show it as `hdc`. In addition, each hard drive can have up to 4 primary partitions (limit of PC x86 architecture) and each primary partition can also have secondary partitions. Each particular partition is identified with a number:
   - First partition: /dev/hda1
   - Second partition: /dev/hda2
   - Third partition: /dev/hda3
   - Fourth partition: /dev/hda4
2. If there is a SCSI or SATA controller these devices are listed as devices sda, sdb, sdc, sdd, sde, sdf, and sdg in the /dev directory. Similarly, partitions on these disks can range from 1 to 16 and are also in the /dev directory.

When a Linux/UNIX system boots, the Kernel requires a "mounted root filesystem". In the most simplest case, which is when the system has only one storage device, the Kernel has to identify which is the device that contains the filesystem (the root / and all its subdirectories) and then, the Kernel has to make this device "usable". In UNIX, this is called "mounting the filesystem".

For example, if we have a single SATA disk with a single partition, it will be named as /dev/sda1. In this case, we say that the device "/dev/sda1" mounts "/". The root of the filesystem "/" is called the mount point. The file /etc/fstab contains the list of devices and their corresponding mount points that are going to be used by the system.

3.5 Disk Usage

A couple of useful commands for viewing disk capacities and usage are `df` and `du`. The command `df` (abbreviation for disk free) is used to display the amount of available disk space of file systems:

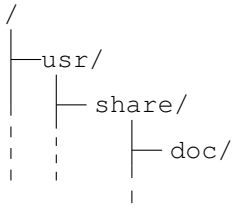
```
$ df -h
Filesystem            Size  Used Avail Use% Mounted on
/dev/sda1             108G   41G   61G  41% /
none                  1,5G  728K  1,5G   1% /dev
none                  1,5G  6,1M  1,5G   1% /dev/shm
none                  1,5G  116K  1,5G   1% /var/run
none                  1,5G     0  1,5G   0% /var/lock
```

`du` (abbreviated from disk usage) is a used to estimate file space used under a particular directory or by certain files on a file system:

```
$ du -sh /etc/apache2/
464K    /etc/apache2/
```

3.6 The path

In Unix-like systems the filesystem has a root denoted as **/** (do not get confused with the *root* user). All the files on the system are named taking the FS root as reference. In general, the *path* defines how to reach a file in the FS. For instance, */usr/share/doc* points out that the file *doc* (which is a directory) is inside the directory *share* which is inside the directory *usr*, which is under the FS root (*/*).



We have three basic commands that let us move around the FS and list its contents:

- The command `ls` (list) lists the files on the current directory.
- The command `cd` (change directory) allows us to change from one directory to another.
- The command `pwd` (print current working) prints the current directory.

The directories contain two special names:

. (a dot) which represents the current directory.

.. (two dots) which represent the parent directory.

With commands related to the filesystem you can use absolute and relative names for the files:

- **Absolute path.** An absolute path always takes the root **/** of the filesystem as starting point. Thus, we need to provide the full path from the root to the file. Example: `/usr/local/bin`.
- **Relative path.** A relative path provides the name of a file taking the current working directory as starting point. For relative paths we can use **.** (the dot) and **..** (the two dots). Examples:

  `./Desktop` or for short `Desktop` (the ./ can be omitted). This is names a file called Desktop inside the current directory.

  `./../../etc` or for short `../../etc`. This names the file (directory) etc, which is located two directories up in the FS.

Finally, the special character ~ (**`ALT GR+4`**) can used as the name of your "home directory" (typically `/home/username`). Recall that your home directory is the area of the FS in which you can store your files.

Examples:

```
$ ls /usr
bin  games  include  lib  local  sbin  share  src
$ cd /
$ pwd
/
$ cd /usr/local
$ pwd
  /usr/local
$ cd bin
$ pwd
  /usr/local/bin
$ cd /usr
$ cd ./local/bin
$ pwd
  /usr/local/bin
$ cd ../../share/doc
$ pwd
  /usr/share/doc
$ ls ~
Downloads  Videos Desktop  Music
```

3.7 Directories

In a FS, files and directories can be created, deleted, moved and copied. To create a directory, we can use `mkdir`:

```
$ cd ~
$ mkdir myfolder
```

This will create a directory called "myfolder" inside your working directory (e.g. /home/user). If we want to delete a directory, we can use `rmdir`:

```
$ rmdir ~/myfolder
```

The previous command will fail if the folder is not empty (contains some other file or directory). There are two ways to proceed: delete the content and then the directory itself or force a recursive removal with the command `rm` and the options -r (recursive) and -f (force). Example:

```
$ rm -rf myfolder
```

This is analogous to:

```
$ rm -f -r myfolder
```

The command `mv` (move) can be used to move a directory to another location:

```
$ mkdir folder1
$ mkdir folder2
$ mv folder2 folder1
```

You can also use the command `mv` to rename a directory:

```
$ mv folder1 directory1
```

Finally, to copy the contents of a directory to other place in the file system we can use the `cp` command with the option -r (recursive). Example:

```
$ cd directory1
$ mkdir folder3
$ cd ..
$ cp -r directory1 directory2
```

3.8 Files

The easiest way to create a file is using `touch`:

```
$ touch test.txt
```

This creates an empty file called `test.txt`. To remove this file, the `rm` command can be used:

```
$ rm test.txt
```

Obviously, if a file which is not in the working directory has to be removed, the complete path must be the argument of `rm`:

```
$ rm /home/user1/test.txt
```

In order to move or rename files we can use the `mv` command. For example, moving the file test.txt to the Desktop directory on your home might look like this:

```
$ mv test.txt ~/Desktop/
```

In case a name is specified in the destination, the resulting file will be renamed:

```
$ mv test.txt Desktop/test2.txt
```

Renaming a file can be done also with `mv`:

```
$ mv test.txt test2.txt
```

The copy command works similar to `mv` but the origin will not disappear after the copy operation:

```
$ cp test.txt test2.txt
```

Finally, we have hidden files. Ah hidden file is any file that begins with a ".". When a file is hidden it can not been seen with the bare `ls` command or an un-configured file manager. In most cases you won't need to see those hidden files as much of them are configuration files/directories for your desktop. There are times, however, that you will need to see them in order to edit them or even navigate through the directory structure. To do this you will need to use the option -a with `ls`:

```
$ ls -a
```

3.9 PATH variable

You may wonder how the system knows the path to commands because normally we do not type any relative or absolute path to the command but just the command name. The response is that the system utilizes the environment variable PATH. You can check the contents of PATH as:

```
$ echo $PATH
```

3.10 File content

It is usual to end the names of files with an extension, which is a dot followed by some characters. The extension is used to provide some hint about the content of a file. Examples: .txt for text files , .jpg or .jpeg for jpeg images, .htm or .html for HTML documents, etc. However, in Unix systems, the file extension is optional. If a file does not have a extension and you want to know which is its content, GNU/Linux provides you with a command that uses a guessing mechanism called *magic numbers*. The command is `file`. Example:

```
$ file /etc/services
/etc/services: ASCII English text
```

3.11 File expansions and quoting

Bash provides us with some special characters that can be used to name groups of files. This special characters have a special behavior called "filename expansion" when used as names of files. We have several filename expansions:

Character	**Meaning**
?	Expands one character.
*	Expands zero or more characters (any character).
`[ ]`	Expands one of the characters inside [ ].
!( )	Expands the file expansion **not** inside ( ).

Examples:

```
$ cp ~/* /tmp            # Copies all the files in your home to /tmp
$ cp -r * ~              # Copies everything recursively in the current directory
                         # to your home
$ rm ~/hello?            # Removes files in your home called "hello0" or
                         # "hellou" but not "hello" or "hellokitty".
$ cp ~/[Hh]ello.c /tmp   # Copies Hello.c and hello.c from your home (if they exist)
                         # to /tmp.
$ rm -r !(*.jpg)         # Deletes everything from the current directory recursively
                         # except files in the form *.jpg
```

These special characters for filename expansions can be disabled with quoting:

Character	**Action**
' (simple quote)	All characters between simple quotes are interpreted without any special meaning.
" (double quotes)	Special characters are ignored except `$`, ` ` ' and \
\ (backslash)	The special meaning of the character that follows is ignored.

Example:

```
$ rm "hello?"  # Removes a file called hello?
               # (but not a file called hello1).
```

3.12 Text Files

A text file typically contains human readable characters such as letters, numbers, punctuation, and also control characters such as tabs, line breaks, carrier returns, etc. The simplicity of text files allows a large amount of programs to read and modify it. Text files contain bytes representing characters that must read using a character encoding table or charset. The most well known character encoding table is the ASCII table. The ASCII table defines control and printable characters. The original specification of the ASCII table defined only 7 bits. Examples of 7-bit ASCII codification are:

```
a: 110 0001 (97d) (Ox61)
A: 100 0001 (65d) (Ox41)
```

Later, the ASCII table was expanded to 8 bits (a byte). Examples of 8-bit ASCII codification are:

```
a: 0110 0001
A: 0100 0001
```

As you may observe, to build the 8-bit codification, the 7-bit codification was maintained just setting a 0 before the 7-bit word. For those words whose codification started with 1, several specific encodings per language appeared. These codifications were defined in the ISO/IEC 8859 standard. This standard defines several 8-bit character encodings. The series of standards consists of numbered parts, such as ISO/IEC 8859-1, ISO/IEC 8859-2, etc. There are 15 parts. For instance, ISO/IEC 8859-1 is for Latin languages and includes Spanish and ISO/IEC 8859-7 is for Latin/Greek alphabet. An example of 8859-1 codification is the following:

```
ç (ASCII): 1110 0111 (231d) (Oxe7)
```

Nowadays, we have other types of encodings. The most remarkable is UTF-8 (UCS Transformation Format 8-bits), which is the default text encoding used in Linux. UFT-8 defines a variable length universal character encoding. In UTF-8 characters range from one byte to four bytes. UTF-8 matches up for the first 7 bits of the ASCII table, and then is able to encode up to 2^{31} characters unambiguously (universally). Example:

```
ç (UTF8): Oxc3a7
```

Finally, in a text file we must define how to mark a **new line**. A new line, line break or end-of-line (EOL) is a special character or sequence of characters signifying the end of a line of text. In ASCII (or compatible charsets) the characters LF (Line feed, "`\n`", 0x0A, 10 in decimal) and CR (Carriage return, "`\r`", 0x0D, 13 in decimal) are reserved to mark the end of a text line. The actual codes representing a newline vary across operating systems:

- **CR+LF**: Microsoft Windows, DEC TOPS-10, RT-11 and most other early non-Unix and non-IBM OSes, CP/M, MP/M, DOS (MS-DOS, PC-DOS, etc.), Atari TOS, OS/2, Symbian OS, Palm OS.
- LF+CR: Acorn BBC spooled text output.
- CR:Commodore 8-bit machines, Acorn BBC, TRS-80, Apple II family, Mac OS up to version 9 and OS-9.
- **LF**: Multics, Unix and Unix-like systems (GNU/Linux, AIX, Xenix, Mac OS X, FreeBSD, etc.), BeOS, Amiga, RISC OS, Android and others.

The different codifications for the newline can be a problem when exchanging data between systems with different representations. If for example you open a text file from a windows-like system inside a unix-like system you will need to either convert the newline encoding or use a text editor able of detecting the different formats (like `gedit`).

Note. For text transmission, the standard representation of newlines is CR+LF.

3.13 Commands and Applications for Text

Many applications designed to manipulate text files, called text editors, allow the user to edit and save text with several encodings. For example, a text editor that uses the GUI is "gedit". On the terminal we have also several text editors. The most remarkable one is `vi`[1] or `vim` (a enhanced version of vi). `vi` might be little cryptic but it is useful because it is present in almost any Unix system. Let's get familiarized with `vi`. For example, to start editing the file myfile.txt with `vi`, you should type:

```
$ vi myfile.txt
```

The previous command puts `vi` in *command mode* to edit myfile.txt. In this mode, you can navigate through myfile.txt and quit by typing `:q!`. Also, in this mode you can delete a line with *dd*, delete from the cursor to the end of the line with *d$* and from the cursor to the beginning of the line with *d^*. You can go to a determinate line with *Gn* (where *n* is the number of the line). If you want to edit the file, you have to press "`i`", which puts `vi` in *insertion mode*. After modifying the document, you can hit `ESC` to go back to *command mode* (default one).

To save the file you must type `:wq` and to quit without saving, you must force the exit by typing `:q!`.

On the other hand, there are also other commands to view text files. These commands are `cat`, `more` and `less`. The `less` command works in the same way as `man` does. Try:

```
$ cat /etc/passwd
$ cat /etc/passwd /etc/hostname
$ more /etc/passwd
$ less /etc/passwd
```

Another couple of useful commands are `head` and `tail`, which respectively, show us the text lines at the top of the file or at the bottom of the file.

```
$ head /etc/passwd
$ tail -3 /etc/passwd
```

A very interesting option of tail is -f, which outputs appended data as the file grows. Example:

```
$ tail -f /var/log/syslog
```

If we have a binary file, we can use `hexdump` or `od` to see its contents in hexadecimal and also in other formats. Another useful command is `strings`, which will find and show characters or groups of characters (strings) contained in a binary file. Try:

[1] There are other command-line text editors like `nano`, `joe`, etc.

```
$ hexdump /bin/ls
$ strings /bin/ls
$ cat /bin/ls
```

There are control characters in the ASCII tables that can be present in binary files but that should never appear in a text file. If we accidentally use `cat` over a binary file, the prompt may turn into a strange state. To exit this state, you must type `reset` and hit ENTER.

Other very useful commands are those that allow us to search for a pattern within a file. This is the purpose of the `grep` command. The first argument of `grep` is a pattern and the second is a file. Example:

```
$ grep bash /etc/passwd
$ grep -v bash /etc/passwd
```

Finally, another interesting command is `cut`. This command can be used to split the content of a text line using a specified delimiter. Examples:

```
$ cat /etc/passwd
$ cut -c 1-4 /etc/passwd
$ cut -d ":" -f 1,4 /etc/passwd
$ cut -d ":" -f 1-4 /etc/passwd
```

3.14 Links

A link is a special file. Links can be hard or symbolic (soft).

- **A Hard Link** is a way of giving another name to a file.
  - Each name (hard link) can use a different location in the filesystem.
  - Hard links must refer to existent data in a certain file system.
  - Two hard links offer the same functionality with different names. E.g. any of the hard links can be used to modify the data of the file.
  - A file will not exist anymore if all its hard links are removed.
- **A Symbolic Link (also called Soft Link)** is a new (different) file whose contents are a pointer to another file or directory.
  - If the original file is deleted, the soft link becomes unusable.
  - The soft link is usable again if original file is restored.
  - Soft links allow to link files and directories between different FS, which is not allowed by hard links.

The `ln` command is used to create links. If the -s option is passed as argument, the link will be symbolic. Examples:

```
 $ ln -s /etc/passwd ~/hello
 $ cat ~/hello
```

The previous `ln` command creates a symbolic link to `/etc/passwd` and the `cat` command prints the contents as text of the link. We can also use hard links to rename a file. For example:

```
 $ touch file1.txt
 $ ln file1.txt file2.txt
 $ stat file1.txt
  File: `file1.txt'
  Size: 0               Blocks: 0          IO Block: 4096   regular empty file
Device: 811h/2065d      Inode: 1056276     Links: 2
Access: (0664/-rw-rw-r--)  Uid: ( 1000/ user)   Gid: ( 1000/ user)
Access: 2013-06-25 18:47:52.675904819 +0200
Modify: 2013-06-25 18:47:52.675904819 +0200
```

```
Change: 2013-06-25 18:48:07.667904983 +0200
 Birth: -
 $ rm file1.txt
```

The previous `ln` and `rm` commands are equivalent to the following `mv` command:

```
 $ mv file1.txt file2.txt
```

Links, especially symbolic links, are frequently used in Linux system administration. Commands are often aliased so the user does not have to know a version number for the current command, but can access other versions by longer names if necessary. Library names are also managed extensively using symlinks, for example, to allow programs to link to a general name while getting the current version.

3.15 Unix Filesystem Permission System

Introduction

Unix operating systems are organized in users and groups. Upon entering the system, the user must enter a login name and a password. The login name uniquely identifies the user. While a user is a particular individual who may enter the system, a group represents a set of users that share some characteristics. A user can belong to several groups, but at least the user must belong to one group. The system also uses groups and users to perform some of its internal management tasks. For this reason, in addition to real users, in a Unix system there will be other users. Generally, these "special" users cannot login into the system, i.e., they cannot have a GUI or a CLI.

Permissions

Linux FS provides us with the ability of having a strict control of files and directories. To this respect, we can control which users and which operations are allowed over certain files or directories. To do so, the basic mechanism (despite there are more mechanisms available) is the "Unix Filesystem Permission System".

There are three specific permissions on this permission system:

- The **read permission**, which grants the ability to read a file. When set for a directory, this permission grants the ability to read the names of files in the directory (but not to find out any further information about them such as contents, file type, size, ownership, permissions, etc.)
- The **write permission**, which grants the ability to modify a file. When set for a directory, this permission grants the ability to modify entries in the directory. This includes creating files, deleting files, and renaming files.
- The **execute permission**, which grants the ability to execute a file. This permission must be set for executable binaries (for example, a compiled C++ program) or shell scripts to allow the operating system to run them. When set for a directory, the execution permission grants the ability to traverse the directory to access to files or subdirectories, but it does not grant the permission to view the directory content (unless the read permission is set for the directory).

When a permission is not set, the rights it would grant are denied. Unlike other systems, permissions on a Unix-like system are not inherited. Files created within a directory will not necessarily have the same permissions as that directory. On the other hand, from the point of view of a file, the user is in one of the three following categories or classes:

- User Class. The user is the owner of the file.
- Group Class. The user belongs to the group of the file.
- Other Class. Neither of the two previous situations.

The most common form of showing permissions is symbolic notation. The following are some examples of symbolic notation:

`-rwxr-xr-x` a regular file whose user class has full permissions and whose group and others classes have only the read and execute permissions.

`dr-x------` a directory whose user class has read and execute permissions and whose group and others classes have no permissions.

The command to list the permissions of a file is `ls -l`. Example:

```
$ls -l /usr
total 188
drwxr-xr-x   2 root root 69632 2011-08-23 18:39 bin
drwxr-xr-x   2 root root  4096 2011-04-26 00:57 games
drwxr-xr-x  41 root root  4096 2011-06-04 02:32 include
drwxr-xr-x 251 root root 69632 2011-08-20 17:59 lib
drwxr-xr-x   3 root root  4096 2011-04-26 00:56 lib64
drwxr-xr-x  10 root root  4096 2011-04-26 00:50 local
drwxr-xr-x   9 root root  4096 2011-06-04 04:11 NX
drwxr-xr-x   2 root root 12288 2011-08-23 18:39 sbin
drwxr-xr-x 370 root root 12288 2011-08-08 08:28 share
drwxrwsr-x  11 root src   4096 2011-08-20 17:59 src
```

Change permissions (chmod)

The command `chmod` is used to change the permissions of a file or directory.

Syntax: `chmod user_type operation permissions file`

User Type	u	user
	g	group
	o	other
Operation	+	Add permission
	-	Remove permission
	=	Assign permission
Permissions	r	reading
	w	writing
	x	execution

For example, to assign the read and execute permissions to the group class of the file ”temp.txt“ we can type the following command:

```
$ chmod g+rx temp.txt
```

Changing the permissions of the file ”file1.c“ for allowing only the user to read its contents can be achieved by:

```
$ chmod u=r file1.c
```

Another example for setting the permissions of the user and the group simultaneously:

```
$ chmod u=r,g=rx file1.c
```

Another way of managing permissions is to use octal notation. With three-digit octal notation, each numeral represents a different component of the permission set: user class, group class, and other class respectively. Each of these digits is the sum of its component bits. Here is a summary of the meanings for individual octal digit values:

```
0 --- no permission
1 --x execute
2 -w- write
3 -wx write and execute
```

```
4 r-- read
5 r-x read and execute
6 rw- read and write
7 rwx read, write and execute
```

For example, to grant the read permission for all the classes, the write permission only for the user and execution for the group over the file "file1.c" you should type:

```
$  chmod 654 file1.c
```

The numeric values come from:

$$r_{user} + w_{user} + r_{group} + x_{group} + r_{other} = 400 + 200 + 40 + 10 + 4 = 654$$

Default permissions

Users can also establish the default permissions for their new created files. The `umask` command allows to define these default permissions. When used without parameters, returns the current mask value:

```
$ umask
0022
```

You can also set a mask. Example:

```
$ umask 0044
```

For security reasons, only read and write permissions can be used by the default mask but not execute. The mask tells us in fact which permission is subtracted (i.e. it is not granted).

3.16 Extra

3.16.1 *inodes

The inode (index node) is a fundamental concept in the Linux and UNIX filesystem. Each object in the filesystem is represented by an inode. Each and every file under Linux (and UNIX) has following attributes:

- Inode number.
- File type (regular file, block special, etc).
- Permissions (read, write etc).
- Owner.
- Group.
- File Size.
- Access Time (atime). This is the time that the file was last accessed, read or written to.
- Modify Time (mtime). This is the time that any inode information was last modified.
- Change Time (ctime). This is the last time the actual contents of the file were last modified.
- Number of links (soft/hard).
- Etc.

All the above information stored in an inode. In short the inode identifies the file and its attributes (as above). Each inode is identified by a unique inode number within the file system. Inode is also know as index number. An inode is a data structure on a traditional Unix-style file system such as UFS or ext4. An inode stores basic information about a regular file, directory, or other file system object. You can use `ls -i` command to see inode number of file:

```
$ ls -i /etc/passwd
32820 /etc/passwd
```

You can also use `stat` command to find out inode number and its attributes:

```
$ stat /etc/passwd

File: `/etc/passwd'
Size: 1988            Blocks: 8          IO Block: 4096   regular file
Device: 341h/833d       Inode: 32820       Links: 1
........
```

Many commands often give inode numbers to designate a file. Let us see he practical application of inode number. Let us try to delete file using inode number.

Create a hard to delete file name:

```
$ cd /tmp
$ touch "\+Xy \+\8"
$ ls
```

Try to remove this file with rm command:

```
$ rm \+Xy \+\8
```

Remove file by an inode number, but first find out the file inode number:

```
$ ls -il
```

The `rm` command cannot directly remove a file by its inode number, but we can use `find` command to delete file by inode.

```
$ find . -inum 471257 -exec rm -i {} \;
```

In this case, 471257 is the inode number that we want to delete.

Note you can also use add character before special character in filename to remove it directly so the command would be:

```
$ rm "\+Xy \+\8"
```

If you have file like name like name “2011/8/31” then no UNIX or Linux command can delete this file by name. Only method to delete such file is delete file by an inode number. Linux or UNIX never allows creating filename like this but if you are using NFS from MAC OS or Windows then it is possible to create a such file.

3.17 Command summary

The table 3.1 summarizes the commands used within this section.

3.18 Practices

Exercise 3.1– This exercise is related to the Linux filesystem and its basic permission system.

1. Open a terminal and navigate to your home directory (type `cd ~` or simply `cd`). Then, type the command that using a relative path changes your location into the directory /etc.
2. Type the command to return to home directory using an absolute path.
3. Once at your home directory, type a command to copy the file /etc/passwd in your working directory using only relative paths.
4. Create six directories named: dirA1, dirA2, dirB1, dirB2, dirC1 and dirC2 inside your home directory. You can do this with the command:

Table 3.1: Summary of commands related to the filesystem.

stat	shows file metadata.
file	guess file contents.
df	display the amount of available disk space of a filesystem.
du	file space used under a particular directory or by files of a filesystem.
cd	changes working directory.
ls	lists a directory.
pwd	prints working directory.
mkdir	makes a directory.
rmdir	removes a directory.
rm	removes a file or directory.
mv	moves a file or directory.
cp	copies a file or directory.
touch	updates temporal stamps and creates files.
gedit	graphical application to edit text.
vi	application to edit text from the terminal.
cat	shows text files.
more	shows text files with paging.
less	shows text files like `man`.
head	prints the top lines of a text file.
tail	prints the bottom lines of a text file.
hexdump and `od`	shows file data in hex and other formats.
`strings`	looks for character strings in binary files.
grep	prints text lines that match a pattern.
cut	cuts a text string.
ln	creates hard and soft links.
`chmod`	changes file permissions.
`umask`	shows/sets default access mask of new files.

```
$ mkdir dirA1 dirA2 dirB1 dirB2 dirC1 dirC2
```

Or using a functionality called "brace expansion":

```
$ mkdir dir{A,B,C}{1,2}
```

Then, write a command to delete dirA1, dirA2, dirB1 and dirB2 but not dirC1 or dirC2.

Are you able to find another command that produces the same result?

5. Delete directories dirC2 and dirC1 using the wildcard "?".

6. Create an empty file in your working directory called `temp`.

7. Type a command for viewing text to display the contents of the file, which obviously must be empty.

8. Type a command to display the file metadata and properties (creation date, modification date, last access date, inode etc.).

9. What kind of content is shown for the `temp`? and what kind basic file is?

10. Change to your working directory. From there, type a command to try to copy the file `temp` to the `/usr` directory. What happened and why?

11. Create a directory called `practices` inside your home. Inside `practices`, create two directories called `with_permission` and `without_permission`. Then, remove your own permission to write into the directory `without_permission`.

12. Try to copy the `temp` file to the directories `with_permission` and `without_permission`. Explain what has happened in each case and why.

13. Figure out which is the minimum set of permissions (read, write, execute) that the owner has to have to execute the following commands:

Commands	read	write	execute
`cd without_permission`			
`cd without_permission; ls -l`			
`cp temp ~/practices/without_permission`			

Exercise 3.2– This exercise presents practices about text files and special files.

1. Create a file called orig.txt with the `touch` command and use the command `ln` to create a symbolic link to orig.txt called link.txt. Open the `vi` text editor and modify the file orig.txt entering some text.
2. Use the command `cat` to view link.txt. What can you observe? why?.
3. Repeat previous two steps but this time modifying first the link.txt file and then viewing the orig.txt file. Discuss the results.
4. Remove all permissions from orig.txt and try to modify the link.txt file. What happened?
5. Give back the write permission to orig.txt. Then, try to remove the write permission to link.txt. Type `ls -l` and discuss the results.
6. Delete the file orig.txt and try to display the contents of link.txt with the `cat` command. Then, in a terminal (t1) edit orig.txt with the command:

   ```
   t1$ vi orig.txt
   ```

   While the editor is opened, in anohter terminal (t2) type:

   ```
   t2$ echo hello > link.txt
   ```

   Close `vi` and comment what has happened in this case.
7. Use the command `stat` to see the number of links that orig.txt and link.txt have.
8. Now create a hard link for the orig.txt file called hard.txt. Then, using the command `stat` figure out the number of "Links" of orig.txt and hard.txt.
9. Delete the file orig.txt and try to modify with `vi` hard.txt. What happened?
10. Use the `grep` command to find all the information about the HTTP protocol present in the file /etc/services (remember that Unix commands are case-sensitive).
11. Use the `cut` command over the file /etc/group to display the name of each group and its members (last field).
12. Create an empty file called text1.txt. Use text editor `vi` abn to introduce "abñ" in the file, save and exit. Type a command to figure out the type of content of the file.
13. Search in the Web the hexadecimal encoding of the letter "ñ" in ISO-8859-15 and UTF8. Use the command `hexdump` to view the content in hexadecimal of text1.txt. Which encoding have you found?
14. Find out what the character is "0x0a", which also appears in the file.
15. Open the `gedit` text editor and type "abñ". Go to the menu and use the option "Save As" to save the file with the name text2.txt and "Line Ending" type Windows. Again with the `hexdump` examine the contents of the file. Find out which is the character encoded as "0x0d".

```
$ hexdump text2.txt
0000000 6261 b1c3 0a0d
0000006
```

16. Explain the different types of line breaks for Unix (new Mac), Windows and classical Mac.

17. Open the gedit text editor and type ”abñ“. Go to the menu and use the option “Save As” to save the file with the name text3.txt and “Character Encoding” ISO-8859-15. Recheck the contents of the text file with `hexdump` and discuss the results.

Chapter 4

File Descriptors

4.1 File Descriptors

A file descriptor (fd) is an abstract indicator for accessing a file. More specifically, a file descriptor is an integer that is used as index for an entry in a kernel-resident data structure containing the details of all open files. In Unix-like systems this data structure is called file descriptor table, and **each process has its own file descriptor table**.

The user application passes the fd to the kernel through a system call, and the kernel will access the file on behalf of the application, based on the fd. The application itself cannot read or write the file descriptor table directly. The same file can have different file descriptors because in fact, each file descriptor refers to a certain access to the file. For example, the same file might be opened only for reading and for this purpose we could obtain a certain *fd*, while we can also open the same file only for writing and obtain for this purpose another *fd* (of course, a file can also be opened for both read and write).

On the other hand, one of the main elements contained in the file descriptor table is the file pointer. The file pointer contains the current position in the file for reading or writing. So, we can have the same file opened twice for reading but with the file pointer associated with each *fd* placed at a different position. The *open*[1] system call is used to access files and get the corresponding *fd*. This system call takes among others as parameters the file's pathname and the kind of access requested on the file (read, write, append etc.). Once the file is opened, we can use other system calls to read, write, close, position the file pointer, etc. There are 3 standard file descriptors which presumably every process should have opened (see Table 4.1).

Table 4.1: *Standard File Descriptors*

fd (integer value)	**Name**
0	Standard Input (STDIN)
1	Standard Output (STDOUT)
2	Standard Error (STDERR)

When a process is generated using a shell (like Bash), it inherits the 3 standard file descriptors from this shell. Let's open a terminal and discover which are these three "mysterious" file descriptors. First, let's discover the PID of the bash:

```
$ ps
 PID TTY          TIME CMD
14283 pts/3    00:00:00 bash
14303 pts/3    00:00:00 ps
```

The command `lsof` ("list open files") will help us to find which are these files.

[1] In Unix-like systems, file descriptors can refer to regular files or directories, but also to block or character devices (also called "special files"), sockets, FIFOs (also called named pipes), or unnamed pipes. In what follows we will explain which are these other types of files.

```
$ lsof -a -p 14283 -d0-10
COMMAND  PID    USER     FD  TYPE DEVICE SIZE NODE NAME
bash    14283 telematics 0r  CHR  136,3  5    /dev/pts/3
bash    14283 telematics 1w  CHR  136,3  5    /dev/pts/3
bash    14283 telematics 2w  CHR  136,3  5    /dev/pts/3
```

As we can see in the RD column, the file descriptors for 0,1 and 2 are connected to the file /dev/pts/3. That is to say, to the TTY attached to the terminal. Recall that we can find the TTY of a terminal with the `tty` command and find that it is a special file with the `file` command:

```
$ tty
/dev/pts/3
$ file /dev/pts/3
/dev/pts/3: character special
$ ls -l /dev/pts/3  # Another way of see that /dev/pts/3 is a special file
crw--w---- 1 telematics tty 136, 3 2013-03-03 15:18 /dev/pts/3
```

Another interesting command is the command `fuser`. Among other functionality, providing a file as parameter, `fuser` can show the PID of the processes that have currently open this file. Example:

```
$ fuser /dev/pts/3
/dev/pts/3:           14283
```

The TTY file and the file descriptors 0,1 and 2 are the way in which we can exchange text between the user and the command interpreter of the system (shell). For the previous shell (bash), we can observe that fd=0 is opened for reading (0r) and fd=1,2 are opened for writing (1w and 2w). The fd=0 of any process also receives the name **standard input** or **STDIN**. In this case, the STDIN is used to read the text typed by the user in the terminal and send it to the bash. On the other hand, programs executed from a certain shell inherit by default the open fds of that bash. This includes fd=0 and also fd=1 and fd=2. These latter fds are called respectively **standard output (STDOUT)** and **standard error (STDERR)** (see Figure 4.1).

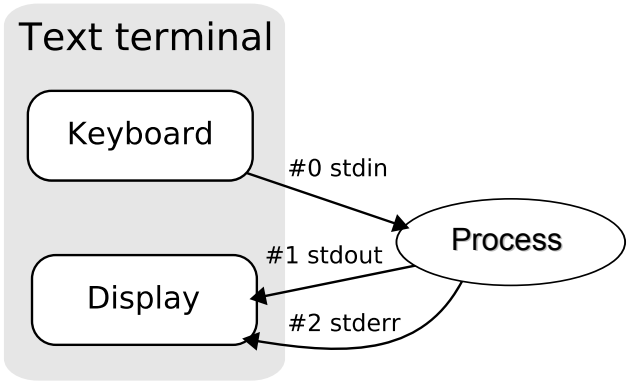


Figure 4.1: Standard Descriptors

A process should write its ”normal“ output to fd=1 and its ”error“ output to fd=2. It is interesting to use two different fds because this allows us to provide a differentiated treatment for normal output and error output, for example, when redirection is used as we explain in the next section.

4.2 Redirecting Output

The “output redirection” is a feature that allows us to connect the STDOUT or STDERR of a process with files different from the default one (the terminal TTY). We will show this functionality through a few examples.

```
$ echo Hello, how are you?
Hello, how are you?
```

As shown, the `echo` command displays "text" in the terminal. In more detail, echo writes "text" to fd=1, which is connected by the default to the TTY of the terminal. Using > we can change this behavior. Let's see an example:

```
$ echo Hello, how are you? > file.txt
$ cat file.txt
Hello, how are you?
```

The redirection operator > is used to connect fd=1 to file.txt instead to the TTY. For this reason we do not see any text in the terminal but the output has been saved in file.txt. Put in another words, the fd=1 is not inherited by the echo command but the bash connects this descriptor to the specified file.

On the other hand, using the operator >> we can also redirect STDOUT but in this case, in append mode. Append mode positions the file pointer at the end of the file and therefore, previous contents are not deleted, new text written to the file is added at the end. Example:

```
$ echo Are you ok? >> file.txt
$ cat file.txt
Hello, how are you?
Are you ok?
```

We can also redirect the STDERR (fd=2). Example:

```
$ ls -qw
ls: option requires an argument -- 'w'
Try `ls --help' for more information.
$ ls -qw 2> error.txt      # Now error is not displayed in the terminal
```

In general:

- **N>file**. It is used to redirect fd=N to a file. If the file exists, is deleted and overwritten. In case file does not exist, it is created.
- **N>>file**. Similar to the first case but opens file in mode append.
- & >**file**. Redirects STDOUT and STDERR to file.

Examples:

```
$ LOGFILE=script.log
$ echo "This sentence is added to $LOGFILE" 1> $LOGFILE
$ echo "This statement is appended to $LOGFILE" 1>> $LOGFILE
$ echo "This phrase does not appear in $LOGFILE as it goes to STDOUT."
$ ls /usr/tmp/notexists >ls.txt 2>ls.err
$ ls /usr/tmp/notexists &>ls.all
```

Note that the redirection commands are initialized or "reseted" after executing each command line. Finally, it is worth to mention that the special file /dev/null is used to discard data. example:

```
$ echo hello > /dev/null
```

The output of above command is sent to /dev/null which is equivalent to discard this output.

4.3 Redirecting Input

Basic redirection

Input redirection allows you to specify a file for STDIN (*fd=0*). If we redirect the STDIN for a command, the text input for command will not come from the TTY (text typed in the terminal by the user) but from the specified file. The input redirection operator is <file. We are going to use a script for illustrating the input redirection.

```
#!/bin/bash
# Our third script, using read for fun
echo Please, type a sentence and hit ENTER
read TEXT
echo You typed: $TEXT
```

To redirect the input:

```
$ echo hello world > file.txt
$ ./thirdscript.sh < file.txt
Please, type a sentence and hit ENTER
You typed: hello world
$
```

As you can observe, the script now reads its input from file.txt instead of the terminal's TTY. We can modify our previous script to show additional information like the PID of the script (cloned bash for executing the script commands), the PID of the script's father or PPID (which is the bash associated to the terminal) and the table of open file descriptors:

```
#!/bin/bash
# Fourth script
echo Script PID: $$
echo Script PPID: $PPID
echo Script (cloned bash) Open Files:
lsof -a -p $$  -d0-2
echo Father bash Open Files:
lsof -a -p $PPID  -d0-2
echo Please, type a sentence and hit ENTER
read TEXT
echo You typed: $TEXT
```

Note. The special variable called "$" contains the PID of the current process.

```
$ ./fourthscript.sh < file.txt
Script PID: 19374
Script PPID: 19294
Script (cloned bash) Open Files:
COMMAND     PID    USER    FD   TYPE DEVICE SIZE/OFF NODE NAME
fourthscr 19374 telematics 0u   CHR  136,1   0t0    4 /home/telematics/file.txt
fourthscr 19374 telematics 1u   CHR  136,1   0t0    4 /dev/pts/1
fourthscr 19374 telematics 2u   CHR  136,1   0t0    4 /dev/pts/1
Father bash Open Files:
COMMAND     PID    USER    FD   TYPE DEVICE SIZE/OFF NODE NAME
fourthscr 19294 telematics 0u   CHR  136,1   0t0    4 /dev/pts/1
fourthscr 19294 telematics 1u   CHR  136,1   0t0    4 /dev/pts/1
fourthscr 19294 telematics 2u   CHR  136,1   0t0    4 /dev/pts/1
Please, type a sentence and hit ENTER
hello world
You typed: hello world
```

Here Documents

Another input redirection is based on internal documents or " here documents".

```
$ cat <<<'hello world'
hello world
$ cat <<END
> hello world
> cruel
> END
hello world
cruel
```

A here document is essentially a temporary file. The redirection operators for here documents are "<<<" and <<EXPR. In the latter input redirection operator you can type text until you enter "EXPR". The first command creates a temporary file with the content hello word. The second command does something similar but you can entry text until you type END.

4.4 Unnamed Pipes

Unix-based operating systems offer, with a redirection operator, a unique approach to join the execution of two or more commands on the terminal. This redirection operator is the "pipe" (symbol "|"). A pipe connects the STDOUT of a command with the sdtin of another command. Communication between different programs allow implementing complex tasks and offer infinite possibilities. Example:

```
$ ls | grep x
```

In the previous example, `ls` produces a list with the contents of the current directory, then `grep` prints only those lines containing the letter "x". In more detail, this type of pipe is called "unnamed pipe" because the pipe exists only inside the kernel and cannot be accessed by processes that created it.

All Unix-like systems include a variety of commands to manipulate text outputs. We have already seen some of these commands: `head`, `tail`, `grep` and `cut`. We also have other commands like `uniq` which displays or removes repeating lines, `sort` which lists the contents of the file ordered alphabetically or numerically, `wc` which counts lines, words and characters and `find` which searches for files. The following example shows a compound command with several pipes (also called "pipeline"):

```
$ cat *.txt | sort | uniq > result.txt
```

The above command line outputs the contents of all the files ending with .txt in the current directory but removing duplicate lines and alphabetically sorting these lines. The result is saved in the file result.txt.

Another useful filter-related command is `tee`. This command is normally used to split the output of a program so that it can be seen on the display terminal and also be saved in a file. The command can also be used to capture an intermediate output before the data is altered by another command. The `tee` command reads standard input, then writes its content to standard output and simultaneously copies it into the specified file(s). The following example lists the contents of the current directory, leaves a copy of these contents in a file called output.txt and then displays these contents in the terminal in reverse order:

```
$ ls | tee output.txt | sort -r
```

Finally, it is worth to mention that when a command line with pipes or pipeline is executed in background, all commands executed in the pipeline are considered members of the same task or job. In the following example, `tail` and `grep` belong to the same task:

```
$ tail -f file.txt | grep hi &
[1] 15789
```

The PID shown, 15789, corresponds to the process ID of the last command of the pipeline (`grep` in this example) and the JID in this case is "1". Signals received by any process of the pipeline affects all the processes of the pipeline.

4.5 Process Substitution

When you enclose several commands in parenthesis, the commands are actually run in a "subshell"; that is, the bash clones itself and is the cloned bash who interprets the commands within the parenthesis (this is the same behavior as with shell scripts). Since the outer shell is running only a "single command", the output of a complete set of commands can be redirected as a unit. For example, the following command writes the list of processes and also the current directory listing to the file commands.out:

```
$ (ps ; ls) >commands.out
```

Process substitution occurs when you put a "<" or ">" in front of the left parenthesis. For instance:

```
$ cat <(ls -l)
```

The previous command-line results in the command `ls -l` executing in a subshell as usual, but redirects the output to a temporary named pipe, which bash creates, names and later deletes. Therefore, `cat` has a valid file name to read from, and we see the output of `ls -l`, taking one more step than usual to do so. Similarly, giving ">(commands)" results in bash naming a temporary pipe, which the commands inside the parenthesis read for input. Process substitution also makes the `tee` command more useful in that you can cause a single input to be read by multiple readers without resorting to temporary files. With process substitution bash does all the work for you. For instance:

```
ls|tee >(grep foo | wc>foo.count) | tee >(grep bar | wc>bar.count) | grep baz | wc>baz.count
```

The previous command-line counts the number of occurrences of foo, bar and baz in the output of `ls` and writes this information to three separate files.

4.6 Dash

The dash "-" in some commands is useful to indicate that we are going to use stdin or stdout instead of a regular file. An example of such type of command is `diff`. To show how does dash works, we will generate two files. The first file called doc1.txt has to contain eight text lines and four of these text lines must contain the word "linux". The second file called doc2.txt has to contain the four lines containing the word "linux" that we introduced in the file doc1.txt. The following command compares the lines of both files that contain the word "linux" and checks that these lines are the same.

```
$ grep linux doc1.txt | diff doc2.txt -
```

In the above command, the dash in `diff` replaces stdin, which is connected by the pipe to the output of `grep`.

Note. In general the use of the dash depends on the context in which it is used. The dash, in fact, has other uses. To give an example, the command `cd -` means go to the previous directory visited (nothing to do with redirection).

4.7 Named Pipes

The other sort of pipe is the "named pipe", also called FIFO (First In First Out). FIFO is a method of processing and retrieving data. In a FIFO, the first bytes entered are the first ones to be removed. In other words, the bytes are removed in the same order they are entered. The named pipe is actually a file in the filesystem. On old Linux systems, named pipes are created by the `mknod` command. On more modern systems, `mkfifo` is the standard command for creation of named pipes. Pipes are shown by `ls` as any other file with a couple of differences:

```
$ mkfifo fifo1
$ ls -l fifo1
prw-r--r--   1 telematics  someusers    0 Jan 22 23:11 fifo1
$ file pipe1
pipe1: fifo (named pipe)
```

The "p" in the leftmost column indicates that fifo1 is a pipe. The simplest way to show how named pipes work is with an example. Suppose we have created pipe as shown above. In a pseudo-terminal type:

```
t1$ echo hello how are you? > pipe1
```

In another pseudo-terminal type:

```
t2$ cat < pipe
```

As you will observe, the output of the command run on the first pseudo-terminal shows up on the second pseudo-terminal. Note that the order in which you run the commands does not matter. If you watch closely, you will notice that the first command you run appears to hang. This happens because the other end of the pipe is not yet connected, so the kernel suspends the first process until the second process opens the pipe. In Unix jargon, the process is said to be ”blocked“, since it is waiting for something to happen. The main application of named pipes is to allow communication between different processes without the need to include these programs on the same pipeline, as it is required by unnamed pipes.

4.8 Extra

In this section we explain some useful commands and features that are typically used together with file redirections like pipelines.

4.8.1 *fd in Bash

With bash, we can manage a total of ten file descriptors. 0 is the standard input, 1 is the standard output, 2 is the standard error and there are seven additional fds that take values from 3 to 9. One of the occasions when it is necessary to have more descriptors than just the three standard ones is when we need to permanently redirect output or input. For example, imagine that we want our script to send stdout to a file. For this purpose, we could create a script like the following:

```
1 #!/bin/bash
2 LOGFILE=/var/log/script.log
3 comand1 >$LOGFILE
4 comand2 >$LOGFILE
5 ...
```

As shown, we have to redirect the output for each command line. A way of improving the previous script is to permanently assign the standard output of the script to the corresponding file. For this functionality, we have to use `exec`. Table 4.2 summarizes the functions of `exec`.

Table 4.2: Standard File Descriptors

Syntax	**Meaning**
`exec fd> file`	open file for writing and assign it to `fd`.
`exec fd>>file`	open file for appending and assign it to `fd`.
`exec fd< file`	open file for reading and assign it to `fd`.
`exec fd<> file`	open file for reading/writing and assign it to `fd`.
`exec fd1>&fd2`	open `fd2` for reading as a copy of `fd1`.
`exec fd1<&fd2`	open `fd2` for writing as a copy of `fd1`.
`exec fd>&-`	close `fd`.
`command >&fd`	write **stdout** to `fd`.
`command 2>&fd`	write **stderr** to `fd`.
`command <&fd`	read **stdin** from `fd`.
`command fd1>&fd2`	write `fd1` to `fd2`.

The previous commands are explained below by examples.

Opening in Write Mode

Let us illustrate the use of `exec` for reading files by an example:

```
#!/bin/bash
LOGFILE=logfile.log
exec 1>$LOGFILE
cd /etc; pwd
```

In the previous script we associate stdout (fd=1) to logfile.log with `exec` and therefore we no longer have to redirect the output of each command line. However, if we want to redirect the stdout again to the default file (TTY) at some point, we need to backup the original file descriptor (which probably is assigned to a special terminal file /dev/pts/X). The following example is illustrative:

```
#!/bin/bash
LOGFILE=logfile.log
exec 3>&1                # New fd=3 that is a copy of the current fd=1
exec 1>$LOGFILE
cd /etc; pwd             # stdout goes to the log file
lsof -a -p $$ -d0-10     # We have four fd open for this script
exec 1>&3                # Now fd=1 points the same file as fd=3
cd /usr; pwd             # The stdout goes again to the terminal
```

In another example let us assign the file logfile.log to fd=3 and use this file descriptor number for writing and, finally close the fd. Example:

```
#!/bin/bash
LOGFILE=logfile.log
exec 3>$LOGFILE
lsof -a -p $$ -d0-10 >&3
exec 3>&-
```

Opening in Read Mode

Open a file in read mode works similarly:

```
#!/bin/bash
exec 4<&0
exec <restaurant.txt
while read score type phone
do
  echo $score,$type,$phone
done
exec 0<&4
exec 4>&-
```

Note. You can observe a "while loop" in the script and that the read command is able to store several variables at the same time (text separated by blanks).

The previous script saves the descriptor of stdin (fd=0) in fd=4. Then opens the file restaurants.txt using fd=0. Then reads the file and finally restores fd=0 and closes fd=4.

Opening in R/W Mode

Example:

```
$ echo 1234567890 > numbers.txt
$ exec 3<> numbers.txt
$ read -n 4 <&3          # read 4 characters (moves the file pointer)
$ echo -n . >&3          # write a dot (without LF)
$ exec 3>&-              # close fd=3
$ cat numbers.txt
1234.67890
```

Final Notes

Note 1. You can also do redirections using different fd but only for a single command line (not for all commands executed with the bash). In this case the syntax is the same but we do not use `exec`. Example:

```
$ exec 3>logfile.log                        # Open fd=3
echo "This goes to the logfile" 1>&3     # redirects stdout to fd=1
echo "This goes to stdout"
```

Note 2. Children processes inherit the opened fd of their parent process. The children can close an fd if it is not going to be used.

4.8.2 *tr

The `tr` command (abbreviated from translate or transliterate) is a command in Unix-like operating systems. When executed, the program reads from the standard input and writes to the standard output. It takes as parameters two sets of characters, and replaces occurrences of the characters in the first set with the corresponding elements from the other set. For example, the following command maps 'a' to 'j', 'b' to 'k', 'c' to 'm', and 'd' to 'n'.

```
$ tr 'abcd' 'jkmn'
```

The -d flag causes `tr` to remove characters in its output. For example, to remove all carriage returns from a Windows file, you can type:

```
$ tr -d '\15' < winfile.txt > unixfile.txt
$ tr -d '\r' < winfile.txt > unixfile.txt
```

Notice that CR can be expressed as `\r` or with its ASCII octal value 15. The -s flag causes `tr` to compress sequences of identical adjacent characters in its output to a single token. For example:

```
$ tr -s '\n' '\n' < inputfile.txt >outputfile.txt
```

The previous command replaces sequences of one or more newline characters with a single newline. Note. Most versions of `tr` operate on single byte characters and are not Unicode compliant.

4.8.3 *find

With the `find` command you can find almost anything in your filesystem. In this section (and also in the next one), we show some examples but `find` offers more options. For example, you can easily find all files on your system that were changed in the last five minutes:

```
$ find / -mmin -5 -type f
```

The following command finds all files changed between 5 and 10 minutes ago:

```
$ find / -mmin +5 -mmin -10 -type f
```

+5 means more than 5 minutes ago, and -10 means less than 10. If you want to find directories, use `-type d`. Searching by file extension is easy too. This example searches the current directory for three different types of image files:

```
$ find . -name "*.png" -o -name "*.jpg" -o -name "*.gif" -type f
```

You can also find all files that belong to a specified username:

```
$ find / -user carla
```

Or to a group:

```
$ find / -group admins
```

Review the manual of `find` to see all its possibilities.

4.8.4 *xargs

`xargs` is a command on most Unix-like operating systems used to build and execute command lines from standard input. On many Unix-like kernels[2] arbitrarily long lists of parameters could not be passed to a command. For example, the following command:

```
$ rm /path/*
```

May eventually fail with an error message of "Argument list too long", if there are too many files in /path. The `xargs` command helps us in this situation by breaking the list of arguments into sublists small enough to be acceptable. The command-line below with `xargs` (functionally equivalent to the previous command) will not fail:

```
$ find /path/* -type f | xargs rm
```

In the above command, `find` feeds the input of `xargs` with a long list of file names. `xargs` then splits this list into sublists and calls `rm` once for every sublist. Another example:

```
$ find . -name "*.foo" | xargs grep bar
```

We might have a problem that might cause that the previous commands do not work as expected. The problem arises when there are whitespace characters in the arguments (e.g. filenames). In this case, the command will interpret a single filename as several arguments. In order to avoid this limitation one may use:

```
$ find . -name "*.foo" -print0 | xargs -0 grep bar
```

The above command separates filenames using the NULL character (0x00) instead of using whitespace (0x20) to separate arguments. In this way, as the NULL character is not permitted for filenames we avoid the problem.

4.9 Command summary

Table 4.3 summarizes the commands used within this section.

Table 4.3: Commands related to file descriptors.

`lsof`	displays per process open file descriptors.
`fuser`	displays the list of processes that have opened a certain file.
`uniq`	filter to show only unique text lines.
`sort`	sorts the output.
`wc`	count words, lines or characters.
`diff`	search differences between text files.
`tee`	splits output.
`mkfifo`	creates a named pipe.
`exec`	bash keyword for operations with files.
`tr`	translate text.
`find`	search files.
`xargs`	build argument lines.

4.10 Practices

Exercise 4.1– In this exercise, we will practice with file redirections using several filter commands.

1. Without using any text editor, you have to create a file called mylist.txt in your home directory that contains the recursive list of contents of the /etc directory. Hint: use `ls -R`. Then, "append" the sentence "CONTENTS OF ETC" at the end of the file mylist.txt. Finally, type a command to view the last 10 lines of mylist.txt to check that you obtained the expected result.

[2] Under the Linux kernel before version 2.6.23

2. Without using any text editor, you have to "prepend" the sentence "CONTENTS OF ETC" at the beginning of mylist.txt. You can use auxiliary files but when you achieve the desired result, you have to remove them. Finally, check the result typing a command to view the first 10 lines of mylist.txt.

3. Type a command-line using pipes to count the number of files in the /bin directory.

4. Type a command-line using pipes that shows the list of the first 3 commands in the /bin directory. Then, type another command-line to show this list in reverse alphabetical order.

   Hint: use the commands `ls`, `sort` and `head`.

5. Type the command-lines that achieve the same results but using `tail` instead of `head`.

6. Type a command-line using pipes that shows the "number" of users and groups defined in the system (the sum of both).

   Hint: use the files /etc/passwd and /etc/group.

7. Type a command line using pipes that shows one text line containing the PID and the PPID of the `init` process.

Exercise 4.2– In this exercise, we are going to practice with the special files of pseudo-terminals (/dev/pts/X).

1. Open two pseudo-terminals. In one pseudo-terminal type a command-line to display the the content of the file /etc/passwd in the other terminal.

2. You have to build a chat between two pseudo-terminals. That is to say, what you type in one pseudo-terminal must appear in the other pseudo-terminal and vice-versa.

   Hint: use `cat` and a redirection to the TTY file of the pseudo-terminal.

Exercise 4.3– (*) Explain in detail what happens when you type the following command lines:

```
$ mkfifo pipe1 pipe2
$ echo -n x | cat - pipe1 > pipe2 &
$ cat <pipe2 > pipe1
```

Do you see any output?
Hint. Use `top` in another terminal to see CPU usage.

Exercise 4.4– (*) In this exercise, we deal with inheritance of file descriptors.

1. Execute the command `less /etc/passwd` in two different pseudo-terminals. Then, from a third terminal list all processes that have opened /etc/passwd and check their PIDs.

   Hint: use `lsof`.

2. Using the `fuser` command, kill all processes that have the file /etc/passwd open.

3. Open a pseudo-terminal (**t1**) and create an empty file called file.txt. Open file.txt only for reading with `exec` using fd=4. Create the following script called "openfilescript.sh":

```
1 #!/bin/bash
2 # Scriptname: openfilescript.sh
3 lsof -a -p $$ -d0-10
4 echo "Hello!!"
5 read "TEXT_LINE" <&4
6 echo "$TEXT_LINE"
```

Redirect ”stdout“ permanently (with `exec`) to file.txt in **t1** and explain what happens when you execute the previous script in this terminal. Explain what file descriptors has inherited the child bash that executes the commands of the script.

4. From the second pseudo-terminal (**t2**) remove and create again file.txt. Then, execute “openfilescript.sh” in **t1**. Explain what happened and why.

Part II

Virtualization

Chapter 5

Introduction to Virtualization

5.1 Introduction

Virtualization is a methodology for dividing the resources of a physical computer (**phyhost**) into multiple virtualized operating systems (OS).

5.2 Types of Virtualization

There are several kinds of virtualization techniques for creating a virtual OS. These techniques provide similar features but differ in the degree of abstraction and the methods used [**?**].

- **Hardware Emulation or Virtual machines (VMs)**. This approach allows an hypervisor (**phyhost**) to run an arbitrary guest operating system. The guest OS is not modified and it is not aware that it is not running over real hardware. The main issue with this approach is that some OS instructions require to be in supervisor mode and this causes problems since the guest OS is being executed in the user space of the hypervisor. As a result, we need a virtual machine monitor (VMM) in the hypervisor to analyze executed code and to make it safe on-the-fly. This VMM is part of the "virtualization middleware".

- **Paravirtualization.** In this virtualization approach most of the work of the VMM is implemented in the guest OS code, which is modified to avoid the use of privileged instructions. The paravirtualization technique also enables running different OSs on a single server, but requires them to be ported, i.e. guest kernels must "know" that they are running in a user space of an hypervisor.

- **Virtualization on the OS level, a.k.a. containers virtualization.** This technique shares a kernel for several virtual OS called "containers", where each container has an isolated and secure environment. With containers you can even run different distributions but the kernel is shared among containers.

The previous three techniques differ in complexity of implementation, OS support, level of access to common resources and performance in comparison with an standalone server.

In particular, hardware emulation has a wider scope of usage (many OS), but the poorest performance. Paravirtualization has a better performance than hardware emulation, but can support fewer OSs because these OS have to be modified. Containers virtualization has by far the best performance and scalability compared to the other two, but imposes that all the containers share the same kernel. Figure 5.1 shows a picture of the different virtualization types.

In Figure 5.1, we must point out that regarding containers, we show an example in which the hypervisor kernel is who implements the container technology (this is true for technologies like LXC and Docker). However, there are other historical container technologies (e.g. OpenVZ) which used special kernels over the hypervisor kernel to implement containers.

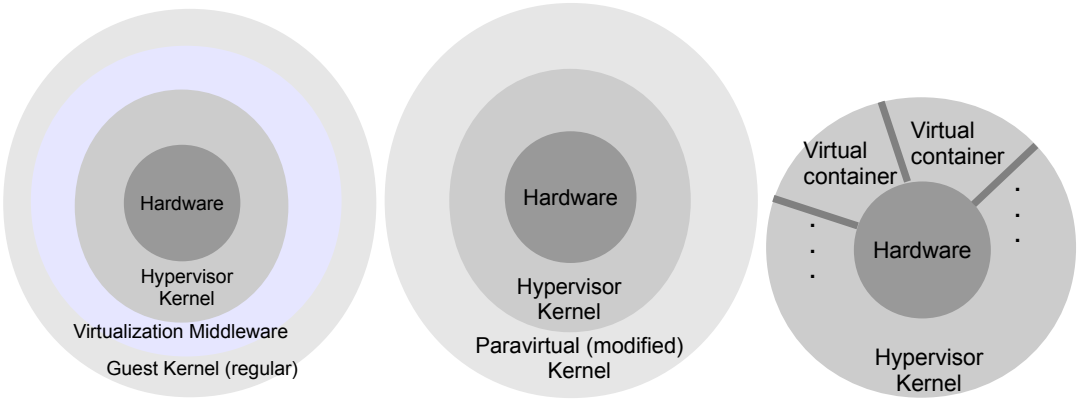


Figure 5.1: Types of Virtualization

5.3 A Virtualized Environment

Virtualized environments have three basic elements (see Figure 5.2):

- **Hypervisor or Physical Host (phyhost).** This is the hardware, the operating system and any other software needed to run the virtual OS (guests).
- **Guests**. These are the virtual OS running over the **phyhost**. A guest might be a traditional OS running just like if it was on a real host. To do so, the host emulates all the system calls for hardware. This makes the guests feel like if they were in a real computer.
- **Virtual switches.** The virtual network is composed of a virtual switches that connect guests like in physical networks. As an additional feature, the **phyhost** can provide connectivity for its guests, allowing them to exchange traffic with other physical networks and even with the Internet.

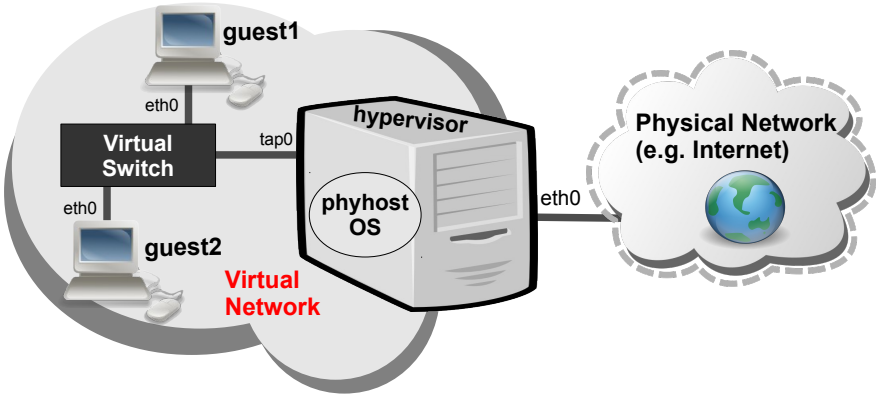


Figure 5.2: Virtualization: a physical host and several guests

The guests are usually accessible with a CLI, a GUI or by some network service like SSH. On the other hand, the **phyhost** can have a virtual network interface that can be connected to the virtual switch. In Linux, we have several types of virtual interfaces like TUN/TAP and Veth.

Chapter 6

User Mode Linux (UML)

6.1 What is UML

User Mode Linux (UML) was created as a kernel development tool for enabling booting a kernel in the user space of another kernel. So if a developer messes with the code and the kernel is unstable, it is not necessary to reboot the **phyhost** kernel, just kill the UML kernel.

However, UML has been used also as a tool for virtualization. In particular, UML is a type of paravirtualization. As such, UML does not require an intermediate virtualization layer or VMM in the **phyhost**. Notice that paravirtualization is less complex than hardware emulation but less flexible too. This is because to run a guest UML Kernel we need a Linux Kernel in the **phyhost** (but a conventional kernel without any modification is enough).

6.2 Building your UML Kernel and Filesystem

To run an UML machine we need a compiled UML kernel and a Linux filesystem. The steps to build these elements are shown next.

UML Kernel

We will be working in the directory `~/uml`. To create it:

```
phyhost:~$ mkdir ~/uml
phyhost:~$ cd ~/uml
phyhost:~/uml$
```

Download[1] and copy your Kernel source code to `~/uml`, then `untar` it and change into the new directory:

```
phyhost:~/uml$ tar -jxvf linux-XXX.tar.bz2
phyhost:~/uml$ cd linux-XXX
phyhost:~/uml/linux-XXX$
```

In this case, the version of the UML kernel that we are going to compile is XXX. Compiling a UML Kernel uses the same procedure as to compiling a standard Kernel, with the exception that every line you type in the process must have the option 'ARCH=um' appended. To compile you need also the package build-essential installed in the **phyhost**. Type the following commands:

```
phyhost:~/uml/linux-XXX$ sudo apt-get install build-essential
phyhost:~/uml/linux-XXX$ make mrproper ARCH=um
phyhost:~/uml/linux-XXX$ make defconfig ARCH=um
phyhost:~/uml/linux-XXX$ make ARCH=um
```

[1] You can download Linux kernels from http://www.kernel.org.

When this completes, you will be have an executable file 'linux' in the ~/uml/linux-XXX/ directory. This is the UML Kernel, adapted to run in a Linux user-space. This kernel is quite big. That is because we have not stripped the debug symbols from it. They may be useful in some cases, but for now we really do not need them so lets remove this debugging info:

```
phyhost:~/uml/linux-XXX$ strip linux
```

The UML Kernel that we have compiled contains the default settings and it is prepared to use modules.

File System for UML

We will show how to create a basic root filesystem that can be used by the UML kernel. With these two elements you will have a fully functional Linux machine, running inside your system (phyhost), but fully isolated and independent. To create the new virtual filesystem, we use the debootstrap command. To install debootstrap, type:

```
phyhost# apt-get install debootstrap
```

Then, we create a 2 GB file to hold the new root filesystem. Create the empty filesystem and format as ext4:

```
phyhost$ cd ~/uml
phyhost:~/uml$ dd bs=1M if=/dev/zero of=debian7.fs count=2048
phyhost:~/uml$ mkfs.ext4 debian7.fs -F
```

Now, create a mount point, and mount this new file so we can begin to fill it up:

```
phyhost:~/uml$ mkdir image
phyhost:~/uml$ sudo mount -o loop debian7.fs image/
```

Use debootstrap to populate this directory with a basic Debian Wheezy:

```
phyhost:~/uml$ sudo debootstrap --arch i386 wheezy image/ ftp://ftp.de.debian.org/debian/
```

The previous command contacts the Debian archive servers and downloads the required packages to get a minimal Debian Wheezy (Debian 7) system installed. Notice I also asked to install vim because it is my preferred command line text editor. Once it completes, if you list the directory image/ you will see a familiar Linux root system.

Kernel Modules

Before running the UML machine we have to install the modules of the kernel into the filesystem. With the image still mounted, type the following commands:

```
phyhost$ cd ~/uml/linux-XXX/
phyhost:~/uml/linux-XXX# make modules_install INSTALL_MOD_PATH=../image ARCH=um
```

The previous commands are equivalent to make a copy of the kernel modules in the directory /lib/modules of the filesystem for the UML machine.

fstab

We must edit the file etc/fstab in the image/ directory to mount the root filesystem when the system boots. Open this file with your preferred text editor (e.g. gedit) and change the contents of the file to the following ones:

```
1 /dev/ubda      /      ext4     defaults     0     1
2 proc           /proc  proc     defaults     0     0
```

Password for *root*

Finally, we must set the password for the *root* user. For this purpose, in a terminal we have to change the root filesystem to the one under the directory `image/`. This is acomplished with the `chroot` command:

```
phyhost:~/uml# chroot image
phyhost# passwd
<type your new UML root password here>
<repeat it>
```

Then to finish the configuration, we exit `chroot` and `umount` the mount point `image/`:

```
phyhost# exit
phyhost:~/uml# umount image
```

Update and Install Soft

To update the system or install a package XXX use the following commands:

```
phyhost# mount -o loop debian7.fs image/
phyhost# cp /etc/resolv.conf image/etc/resolv.conf
phyhost# mount -t proc none image/proc
phyhost# chroot image
```

To install software in the system:

```
phyhost# apt-get update
phyhost# apt-get install XXX  #install package
```

To finish:

```
phyhost# exit
phyhost# umount image/proc
phyhost# fuser -k image
phyhost# umount image
```

6.3 Starting an UML Machine

Let us show you how UML works. Firstly, you have to install the package uml-utilities, which can be installed in the **phyhost** system with the following command:

```
phyhost$ sudo apt-get install uml-utilities
```

Then, let us assume that you have compiled the Linux UML Kernel and that you have created a filesystem (as explained in Section 6.2). It is not absolutely necessary, but let us make a copy of the UML kernel called uml-linux in the directory `uml/`:

```
phyhost:~/uml$ cp linux-XXX/linux uml-linux
```

Then, you can start the UML machine executing the following command:

```
phyhost:~/uml$ ./uml-linux ubda=debian7.fs mem=128M
```

Note. If the previous command does not work go to Section 6.5.

The previous command executes the UML kernel in the user space of the **phyhost** using the filesystem debian7.fs. Notice that we have also specified the size of RAM memory that is going to be used (128 Megabytes). This is a minimal configuration, but UML Kernels support a large number of parameters.

6.4 Copy on Write Filesystems

Now, let us examine how to boot two virtual guests at the same time. We can try to open two terminals and execute the previous command twice but obviously, if kernels try to operate over the same filesystem, we are in trouble because we will have the filesystem in an unpredictable state. A naive solution could be to make a copy of the filesystem in another file and start a couple of UML kernel processes each using a different filesystem file. A better solution is to use the UML technology called COW (Copy-On-Write). COW allows changes to a filesystem to be stored in a file in the **phyhost** separate from the filesystem itself. This has two advantages:

- We can start two UML kernels from the same filesystem.
- Undoing changes to a filesystem is simply a matter of deleting the COW file (the file that contains the changes).

Now, let's fire up our UML kernels with COW. This is achieved basically using the same command line as before, with a couple of changes:

```
phyhost:~uml$ ./uml-linux ubda=cowfile1,debian7.fs mem=128M
```

If the COW file "cowfile1" does not exist, the previous command will create and initialize it.

The COW file is a sparse file. A sparse file is a type of file that attempts to use file system space more efficiently when blocks allocated to the file are mostly empty (which is the case with COW files). This is achieved by writing metadata representing the empty blocks to disk instead of the actual "empty" space which makes up the block, using less disk space. Blocks of the file are written to disk only when they contain "real" (non-empty) data. When reading sparse files, the file system transparently converts metadata representing empty blocks into "real" blocks filled with zero bytes at runtime. The application is unaware of this conversion. You can test with `ls -l` that the size of our COW file is 2GB. On the other hand, the command `du -h` will report the actual "disk usage" occupied on disk (typically Megas).

Once the COW file has been initialized, it can be used alone in the command line:

```
phyhost:~uml$ ./uml-linux ubda=cowfile1 mem=128M
```

The name of the backing file ("debian7.fs") is stored in the COW file header, so it would be redundant to continue specifying it on the command line. The normal way to create a COW file is to specify a non-existent COW file on the UML command line, and let UML create it for you. However, if you want to create a new COW file without booting the UML machine, you can use the `uml_mkcow` command. This command comes with the uml-utilities package. To build a cow file, you can type:

```
phyhost:~uml$ uml_mkcow  cowfile1 debian7.fs
```

Finally, in another terminal we can fire up our second UML kernel with another COW file (cowfile2):

```
phyhost:~uml$ ./uml-linux ubda=cowfile2,debian7.fs mem=128M umid=uml1
```

When you have finished, simply type '`halt`' to stop. You can even '`reboot`' and pretty much anything else without affecting the **phyhost** system in any way.

6.5 Problems and Solutions

To start an UML guest you must be sure that the UML kernel has permission to be executed and that the filesystem has permission to be written. To be sure that these permissions are granted type:

```
phyhost$ chmod u+x uml-linux
phyhost$ chmod u+w debian7.fs
```

If something goes wrong while the UML guest is booting, the Kernel process might go into a bad state. In this case, the best way to "clean" the system is to kill all the processes generated while booting. In our case, as the UML Kernel is called uml-linux, to kill all these processes, we can type the following:

```
phyhost$ killall uml-linux
```

In addition, it might be also necessary to remove the cow files and all the uml related files:

```
phyhost$ rm cowfile?
phyhost$ rm ~/.uml
```

Finally, unless otherwise stated, all the UML programs have to be launched with your unprivileged user (not with the *root* user).

6.6 Networking with UML

6.6.1 Virtual Switch

In this section, we explain how to build a virtual TCP/IP network with UML guests. To build the virtual network we will use the `uml_switch` application (which is in the uml-utilities package). An `uml_switch` can be defined as a virtual switch or a software switch for connection guests. UML instances use internally Ethernet interfaces which are connected to the `uml_switch`. This connection uses a Unix domain socket[2] on the phyhost (see Figure 6.1). In three different terminals (t1,t2 and t3) type the following commands to start two UML machines with COW connected to an `uml_switch`:

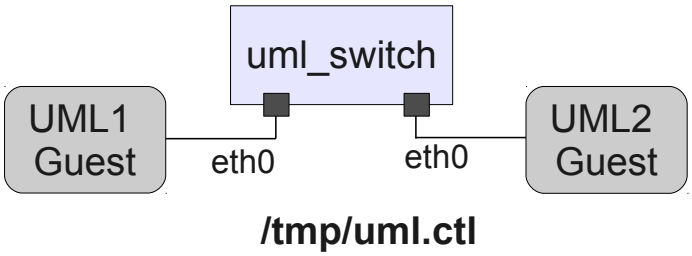


Figure 6.1: Two UML Guests Connected to an `uml_swith`.

```
t1-phyhost$ uml_switch
uml_switch attached to unix socket '/tmp/uml.ctl'
```

```
t2-phyhost:~uml$ ./linux-uml ubda=cowfile1 mem=128M eth0=daemon
```

```
t3-phyhost:~uml$ ./linux-uml ubda=cowfile2 mem=128M eth0=daemon
```

Once you have the two UML guests running, you can enter the usr/passwd (root/xxxx) and configure your IP address and subnet mask using `ifconfig`. For example:

```
UML1$ ifconfig eth0 192.168.0.1 netmask 255.255.255.0
```

```
UML2$ ifconfig eth0 192.168.0.2 netmask 255.255.255.0
```

Then, you can try a `ping` from one UML guest to the other one:

```
UML1$ ping 192.168.0.2
```

[2]Simplifying, a Unix socket is like connecting to a file.

6.6.2 TUN/TAP Interfaces

Tun/Tap devices allow userspace programs to emulate a network device. When the userspace program opens them they get a file descriptor. Packets routed by the kernel networking stack to the device are read from the file descriptor, data the userspace program writes to the file descriptor are injected as local outgoing packets into the networking stack.

The difference between the two types of interfaces is:

- tap sends and receives raw Ethernet frames.
- tun sends and receives raw IP packets.

Once a tun/tap interface is in place, it can be used just like any other interface. IP addresses can be assigned, its traffic can be analyzed, firewall rules can be created, routes pointing to it can be established, etc. We have two ways of creating tun/tap interfaces, which are explained next.

With the `tunctl` Command

One way is using `tunctl` (which is included in the uml-utilities package). This command has to be executed as *root* (or with `sudo`) and you have to indicate which user is going to be able to read/write over this virtual interface.

```
phyhost# tunctl -u telem -t tap0
Set 'tap0' persistent and owned by uid 1000
```

The previous command-line creates a special Ethernet interface called `tap0` and enables the user *telem* to read-/write on `tap0`. Obviously, you must replace *telem* with your unprivileged username.

With the `ip` Command

Another way of adding tun/tap interfaces is to use the `ip` command from iproute2 (iproute2 is the default Linux networking toolkit). To add an tun/tap device useable by root the syntax is:

```
# ip tuntap add dev ${interface name} mode ${mode}
```

Examples:

```
# ip tuntap add dev tun0 mode tun
# ip tuntap add dev tap9 mode tap
```

To add an tun/tap device usable by an ordinary user the syntax is:

```
# ip tuntap add dev ${interface name} mode ${mode} user ${user} group ${group}
```

Examples:

```
# ip tuntap add dev tun1 mode tun user telem group mygroup
# ip tuntap add dev tun2 mode tun user 1000 group 1001
```

6.6.3 Connecting Phyhost

Now, our goal is to enable network communications between the phyhost and the UML guests (see Figure 6.2).

For this purpose, we need to create a tap interface in the phyhost and then, connect this virtual interface to the `uml_swith`:

```
phyhost# tunctl -u telematics -t tap0
Set 'tap0' persistent and owned by uid 1000
```

Then, we can simply run the `uml_swith` (with your unprivileged user) connecting the virtual switch to the special interface `tap0` we have just created. For this, type:

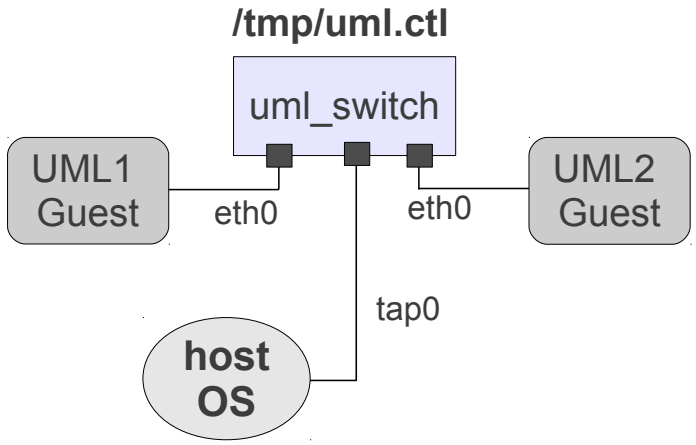


Figure 6.2: Two UML Guests and the Phyhost Connected to an `uml_swith`.

```
phyhost$ uml_switch -tap tap0
uml_switch attached to unix socket '/tmp/uml.ctl' tap device 'tap0'
New connection
```

Now, you can give an IP address and a mask to the `tap0` interface, start the UML guests and try a `ping` from the phyhost to an UML guest:

```
phyhost# ifconfig tap0 192.168.0.3 netmask 255.255.255.0
phyhost# ping 192.168.0.1
```

If everything is configured correctly, the `ping` will succeed.

6.6.4 Multiple Switches

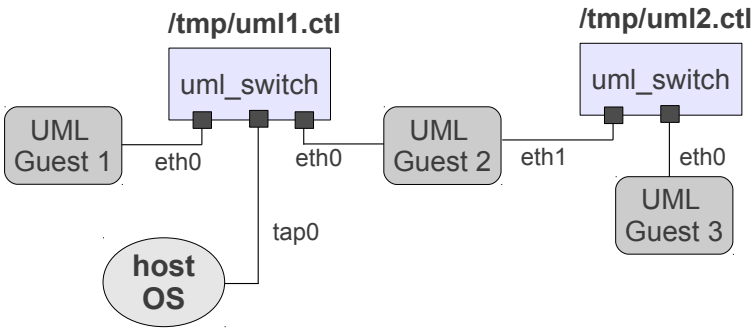


Figure 6.3: UML Topology with 2 Switches.

If you want multiple switches on the host, then the UNIX domain sockets have to be different for each switch. A different socket can be specified with:

```
phyhost$ uml_switch -unix /tmp/uml2.ctl
```

In order to attach to this switch, the same socket must be provided to the UML network driver:

```
phyhost:~uml$ ./linux-uml ubda=cowfile2 mem=128M eth0=daemon,,unix,/tmp/uml2.ctl
```

An Example is shown in Figure 6.3.
Finally, you can attach a UML guest to a tap interface in the phyhost as follows:

```
phyhost:~uml$ ./linux-uml ubda=cowfile2 mem=128M eth0=tuntap,tap0
```

6.7 Practices

Exercise 6.1– Create the topology of the Figure 6.3 starting the guests and `uml_switches` necessary. Configure the guests so that you can do a ping from **Guest1** to **Guest3**.

Chapter 7

Simtools

7.1 Introduction

Simtools is a tool for creating and running virtual environments of guests that use User Mode Linux (UML) kernels. UML is a type of paravirtualization to run Linux guests over an unmodified Linux kernel that serves as hypervisor. Simtools provides several preconfigured scenarios using a modified version of a tool called Virtual Network User Mode Linux (VNUML). VNUML allows us to easily define and run virtual networks using UML Kernels. In particular, VNUML provides us a language and a parser to create these scenarios. In this chapter, we explain how to install simtools and how its main scripts and configuration files work.

7.2 Installation

Currently, simtools is only packaged to be installed in Debian-based Linux systems. In particular, installations have been tested using Ubuntu 12.04/14.04 and Debian 7.

7.2.1 Ubuntu

To install simtools in Ubuntu you have to execute the following commands.

First, to add our repository to your list of APT repositories:

```
wget -O - http://simtools.upc.edu/simtools/keyFile | sudo apt-key add -
echo "deb http://simtools.upc.edu/simtools /"|sudo tee /etc/apt/sources.list.d/simtools.list
```

Then, type the following command to update the software lists:

```
$ sudo apt-get update
```

If your Linux is **a 64-bit system**, install the package libc6:i386

```
$ sudo apt-get install libc6:i386 -y --force-yes
```

Finally, to install simtools:

```
$ sudo apt-get install simtools-meta-PACK -y --force-yes
```

Important: PACK=fxt, sat or tcgi. You can also use PACK=* to install all of them.

You can repeat the previous steps if the software is not installed correctly at the first time, for example, it the network connection is lost.

7.2.2 Debian

There is an issue if you are using Debian. Simtools depends of the mkisofs package but this package has been replaced by genisofs in Debian. However, in the repository of Debian lenny, there is a packet to satisfy this dependency that creates a symlink to genisofs. So, in addition to the previous steps, you have to type the following commands to finish the installation on Debian:

```
$ sudo apt-get install genisofs
$ wget http://archive.kernel.org/debian-archive/debian/pool/main/c/cdrkit/mkisofs_1.1.9-1_all.deb
$ sudo dpkg -i mkisofs_1.1.9-1_all.deb
```

7.2.3 Initial Configuration

Copy the configuration file of screen into your home directory:

```
$ cp /usr/local/share/doc/simtools/screenrc.user ~/.screenrc
```

You can also create a `vnuml` profile on your gnome-terminal (Edit->Profiles->New) with different colors to clearly differentiate the terminals of guests.

7.2.4 Important Directories

There are several important directories for simtools:

- **/usr/share/doc/upc/PACK**. Simtools installs documentation related with a subject and its scenarios in this directory. Remember that PACK=fxt, sat or tcgi.
- **/usr/share/vnuml/kernels**. Simtools stores the UML kernels of the guests in this directory.
- **/usr/share/vnuml/filesystems**. Simtools stores the filesystems of the guests in this directory.
- **/usr/share/vnuml/scenarios**. Simtools stores the definitions of the scenarios of in this directory. By default simtools searches predefined scenarios in this directory. If you want to use another directory, you can set the variable `DIRPRACT` as explained in Section 7.8.2.
- **$HOME/.vnuml**. Simtools stores data about executed scenarios in this directory (local changes).

7.2.5 Update Simtools

You can update the software (scenarios, filesystems, documentation etc.) executing the following commands:

```
$ sudo apt-get update
$ sudo apt-get install simtools-meta-PACK -y --force-yes
```

Where PACK=fxt, sat, tcgi or *

7.2.6 Other Ways of Obtaining Simtools

In **`http://simtools.upc.edu`** we provide updated information about simtools and also other ways of obtaining simtools.

Virtual Machine (OVA)

We also provide to you a **Virtual Machine** with everything already installed on it that can run with hardware virtualization (with VirtualBox or VMWare) . In this case, it is very important that you check that your processor supports hardware virtualization and that you activate this feature in your BIOS. You can check this with the following command:

```
$ grep -E "(vmx|svm)" --color=always /proc/cpuinfo
```

If nothing is displayed after running that command, then your processor does not support hardware virtualization, and you will not be able to use our virtual machine fluently.

USB Stick

Another possibility is to boot from an USB Stick with Ubuntu and simtools installed on it. Currently we have an image for a 16GB pendrive. Then, when booting, select in your system too boot from the USB stick (most modern computers support this disabling UEFI boot). To create the USB stick follow the instructions in our web site.

7.2.7 Remove simtools

If you want to uninstall simtools type:

```
$ sudo apt-get remove simtools*
```

7.3 Start and Stop Scenarios

7.3.1 Start

`simctl` is the main command to manage simtools. If you execute `simctl` without parameters, you will obtain the list of possible scenarios that you can run (by default looking at /usr/share/vnuml/scenarios). It is important to execute `simctl` always with your regular user (never as root). An example output of `simctl` without parameters:

```
phyhost$ simctl
simctl ( dns-basic | fwnat | icmp-class | icmp | ip-routing-abc |
         ip-routing | ip-subnetting | iptunnel | multicast |
         netapps-basic | ospf-basic | rip | switching-vlan | tcp | www ) OPTIONS

OPTIONS
    start                          Starts scenario
    stop                           Stops scenario
    status                         State of the selected simulation
                                   (running | stopped)
    vms                            Shows vm's and tty's from simulation
    labels [vm]                    Shows sequence labels for ALL vm's or for vm
    exec label [vm [vm [ ... ]]]   Exec label in the vm's where label is defined
                                   or exec the label only in the vm list
    netinfo                        Provides info about network connections
    get [-t term_type] vm [pts]    Gets a terminal for vm
                                   term_type selects the terminal
                                   (xterm | gnome | kde | local)
                                   pts is an integer to select the console
    forcestop                      Destroys a simulation scenario
```

The output of `simctl` in this example tells us that it has located several scenarios (dns-basic, fwnat, etc). The previous output also shows us all the possibilities that `simctl` provides us to manage the scenario. These possibilities are explored in the following sections. In addition, you can use the "TAB" key to auto-complete the commands and options available at each moment.

To start a particular scenario, you must execute `simctl` in the **phyhost** with the name of the selected scenario and use the start option. For example:

```
phyhost$ simctl ip-routing-abc start
.......
.......
Total time elapsed: 123 seconds
phyhost$
```

Please, be patient because it might take some time to complete the starting process (this might take up to several minutes). Finally, the command ends indicating the time taken to start the scenario and we get the console prompt again. At this moment, all the virtual machines and their respective interconnections with virtual switches have been created.

After the scenario is started, you can check typing `ifconfig -a` in the **phyhost** that the corresponding tap interfaces (SimNetX) have been created.

7.3.2 Stop

On the other hand, when you wish to stop the simulation, you can type the following:

```
phyhost$ simctl ip-routing-abc stop
.......
Total time elapsed: 17 seconds
phyhost$
```

Usually stopping a simulation is faster than starting it.

Important:

- You **cannot start two simulations at the same time on the same phyhost**.
- Remember to **always stop your simulation before shutting down your phyhost**.

7.3.3 Troubleshooting

A machine might spent some time while booting. Then, it might appear a message telling us to retry, continue or abort. **Type always retry (r)**.

However, if a simulation never starts or stops, to clear the system, type CRL+c and then:

```
phyhost$ simctl simulation_name stop
phyhost$ simctl simulation_name forcestop
```

If your scenarios are not starting, try with:

```
phyhost$ simctl forcestop
```

And reboot **phyhost**.

The `forcestop` option kills all the "linux" processes (UML kernels) and removes the directory .vnuml at the user's home. After rebooting the system you should be able to start the simulation again.

On the other hand, you should never run two different simulations at the same time. If by mistake you start two simulations do:

```
phyhost$ simctl simulation_name1 stop
phyhost$ simctl simulation_name2 stop
phyhost$ simctl forcestop
```

Finally, you should never use the superuser "root" to execute `simctl`. If by mistake you start a simulation with the root user, you must clear the system and start it again using your user:

```
phyhost$ sudo -s
phyhost# simctl simulation_name stop
phyhost# simctl forcestop
phyhost# exit
```

7.4 Access to Virtual Machines

7.4.1 List VMs

One we have a simulation running, we can list the available virtual machines and connect to them using `simctl`. Following our example, let us assume that we have the ip-routing-abc scenario already running. At this moment, we can use the `vm` option to list the virtual machines (UML guests):

```
phyhost$  vms
Virtual machines from ip-routing-abc:
num     vms       enabled tty's Id
  1     alice     0
  2     r1        0 1
  3     r2        0
  4     bob       0
  5     carla     0
```

As we can observe, the machine **r1** has two consoles enabled.

The `get` option of `simctl` allows us to list and access to the virtual machines consoles. If you run the `get` option without parameters, you obtain information about the state of virtual machines ("Running" or "Not Running") and the possibility to access their command consoles.

```
simctl ip-routing-abc get
        alice           Running         --------
        r1              Running         --------
        r2              Running         --------
        bob             Running         --------
        carla           Running         --------
```

In the example shown, the dashed lines (- - - - - - - -) indicate that all virtual machines have enabled consoles but that we have not yet accessed to any of them.

7.4.2 Acess VMs

To access to the console of a virtual machine:

```
phyhost$ simctl ip-routing-abc get alice
```

The `get` option can have an argument (`-t`) to indicate what type of terminal you want to use (it requires that selected terminal emulator is already installed on the system). You can also define a particular terminal as the default for your system in your preferences file "simrc" (see Section 7.8.1).

The terminal can be **xterm** (classic in X11), **gnome** (terminal from GNOME) or **kde** (Konsole from KDE).

For example, to get a gnome-terminal for **bob** virtual machine, you can type:

```
phyhost$ simctl ip-routing-abc get -t gnome bob
```

Once you have accessed to the console of the virtual machines **alice** and **bob**, you can type:

```
phyhost$ simctl ip-routing-abc get
        alice           10681.0         (Attached)
        r1              Running          --------
        r2              Running          --------
        bob             10777.0         (Attached)
        carla           Running          --------
```

"Attached" indicates that there is already a terminal associated with the command console of **alice** virtual machine. If you close the **alice** terminal then the terminal state is "Detached".

Finally, you can also access to the two consoles of **r1** using the following commands:

```
phyhost$ simctl ip-routing-abc get r1 0
phyhost$ simctl ip-routing-abc get r1 1
```

7.5 Network Topology Information

Using the syntax "`simctl simname netinfo`" you can access to the connection topology of virtual machines. For example:

```
phyhost$ simctl ip-routing-abc netinfo
UML             IFACE           NET
HOST            SimNet0         Net0
HOST            SimNet1         Net1
HOST            SimNet2         Net2
HOST            SimNet3         Net3
---------------------------------------
alice           eth0            Net0
bob             eth0            Net3
carla           eth0            Net1
r1              eth0            Net0
r1              eth1            Net2
r1              eth2            Net1
r2              eth0            Net2
r2              eth1            Net3
```

The output is the topology defined in the XML file. This command can be useful to detect mistakes in the topology configuration. It is also worth to mention, that the "netinfo" option uses information directly obtained from the virtual machines (not from the VNUML file) when the simulation is running.

7.6 Managing and Executing Labels

VNUML allows you to define a set of actions to be executed on virtual machines while running a simulation scenario. VNUML uses labels specified in the "seq" attributes of the <exec> and <filetree> tags to associate a label name to a set of actions. Labels allow you to easily distinguish a certain set of actions from another set of actions. The assignment of label names to the associated actions takes place in the VNUML specification file, in particular, in the XML element that defines each virtual machine. The `simctl` wrapper has two options, "labels" and "exec ", to facilitate the management and execution of labels. To show the labels:

```
phyhost$ simctl example-scenario labels
   uml1 : reset_ips
   uml2 : reset_ips enable_forwarding disable_forwarding
   uml3 : reset_ips
```

As shown in the output of the previous command, with the "labels" option we obtain the list of defined labels per virtual machine. You can also view the labels of a specific virtual machine using the name of the virtual machine:

```
phyhost$  simctl example-scenario labels uml2
   uml2 : reset_ips enable_forwarding disable_forwarding
```

The other option of `simctl` ("exec") allows you to manage the execution of the actions (commands) of a label. The command syntax "`simctl simname exec labelname`" executes the commands associated with the label "labelname" on all the virtual machines where that label is defined. For example:

```
phyhost$ simctl example-scenario exec reset_ips
Virtual machines group: uml3 uml2 uml1
OK The command has been started successfully.
OK The command has been started successfully.
OK The command has been started successfully.
Total time elapsed: 0 seconds
```

You can also run a label on a single machine with the following syntax:

```
phyhost$ simctl example-scenario exec disable_forwarding uml2
OK The command has been started successfully.
Total time elapsed: 0 seconds
```

In general, if a label is multiply defined on several virtual machines, with the previous command, the actions of a label are only executed on the specified virtual machine but not on the rest of virtual machines that have defined that label.

Finally, it is worth to mention that `simctl` executes the *start* label when any scenario is started. In addition, many scenarios have a label called *initial*. The *initial* label is executed to load the initial configuration of the scenario. This configuration is stored in your $HOME/.vnuml directory. If you do not have any trouble, you will have to execute initial only once. But if you need to use a `forcestop` option, the initial configuration is removed.

7.7 Capturing Traffic

First of all, it is recommended for using simtools comfortably to enable your unprivileged users for capturing traffic in promiscuous mode. This is accomplished with the following command:

```
$ sudo chmod +s /usr/bin/dumpcap
```

On the other hand, `simctl` automatically creates a `tap` interface in the **phyhost** for each virtual network. These interfaces are called `SimNet0`,`SimNet1`, etc. However, the name (`SimNet`) can be defined as desired (see Section 7.8.1). Simtools also provides a tool called `simtools-captap` to automatically start wireshark protocol analyzers in several `SimNetX` interfaces. The syntax is:

```
phyhost$ simtools-captap
phyhost$ simtools-captap -k  to killall wiresharks
phyhost$ simtools-captap -s 7 to start capturing from SimNet0 to SimNet7
phyhost$ simtools-captap -r 7 to restart capturing from SimNet0 to SimNet7
```

7.8 Tunning Your Config

7.8.1 simrc

`simctl` uses a configuration file for defining some basic parameters using the syntax of `bash`:

```
# simrc: tunning of environment variables for simctl
# Definition of scenario files directory
DIRPRACT=/usr/share/vnuml/scenarios
# Definition of VNUML working directory
# (default to $HOME/.vnuml)
# VNUMLWORKDIR=/tmp/$USER
# Definiton of the user which launch the switch software
TAPUSER=$USER
# Change the default terminal type (xterm)
# values: (gnome | kde | local)
# TERM_TYPE=gnome
# Tap names. For names tap0, tap1 etc. set LEGACY_TAPNAMES=yes.
# With LEGACY_TAPNAMES=no tap names are the $NETPRENAME$NETNAME
# NETNAME is defined in the vnuml file
# For a prename in the tap devices set the NETPRENAME variable.
# LEGACY_TAPNAMES=yes
# NETPRENAME=
# xterm tunning variables
# xterm foreground and background color selection:
# XTERM_FG=white
# XTERM_BG=black
```

```
22 # KDE Konsole tunning
23 # For Konsole version >= 2.3.2 (KDE 4.3.2) use these options:
24 #    KONSOLE_OPTS="-p ColorScheme=GreenOnBlack --title "
25 # For Konsole version <= 1.6.6 (KDE 3.5.10) use these options
26 #      KONSOLE_OPTS="--schema GreenOnBlack -T "
```

To locate this file, the script first checks the file `.simrc` in the home directory of the user that is running `simctl`, if this file is not found, then `simctl` accesses the system-wide configuration file located at `/usr/local/etc/simrc`.

7.8.2 DIRPRACT

To define where to find scenarios in a certain terminal you can redefine the variable DIRPRACT:

```
$ DIRPRACT=mydir
```

7.8.3 Terminals

When `simctl` runs console terminals, it tries to use a classic color settings with green foreground on black background. This feature is modifiable for GNOME and KDE terminals. KDE Konsole terminal can be configured with the variable KONSOLE_OPTS, keeping in mind that the last parameter must be the handle of the window title of the terminal. For gnome-terminal, you can define and save a profile with name *vnuml* with custom features. Editing the gnome-terminal profiles can be made in the edit menu.

7.8.4 Screen

The `screen` application is a screen manager with a VT100/ANSI terminal emulation. `screen` is used by `simctl` to manage the virtual machine terminals through the get function (`simctl simname get vm`) when `pts` option is used in the vnuml file in the <console> tag. However, two problems arise when using pseudo-tty's (pts) with `screen`. While these problems are not critical, they can be annoying. Next, we present the problems and they "workaround" solutions.

- **Problem 1**. The UML Kernel is not aware of terminal size. The consequence of this is that when you resize the terminal in which you are executing `screen`, the size of the terminal is not refreshed.

  Workaround to problem 1. We can use stty command to indicate terminal size to the UML kernel. For example "stty cols 80 rows 24". Elaborating a little bit more this solution, we can use the key binding facility of `screen` to do so.

  - Copy the file /usr/local/share/doc/simtools/screenrc.user into $HOME/.screenrc
  - Put the /usr/local/share/doc/simtools/setgeometry script in a directory included in your PATH environment variable and enable the execution permissions for this script.

  The file .screenrc contains a binding for the key combination **Ctrl+a** and then **f** to the `setgeometry` command. Once in a virtual machine terminal and when the terminal size is modified, keypress **Ctrl+a f** and then the script setgeometry will be executed, finding the new terminal geometry and building, flushing and executing the appropriate `stty` command (`stty cols NUMCOLS rows NUMROWS`) in the virtual machine shell.

- **Problem 2**. The other problem is the lack of screen of terminal scrolling.

  Workaround to problem 2. `screen` has a scrollback history buffer for each virtual terminal of 100 lines high. To enter screen into scrolback mode press **Ctrl+a** and then **ESC**. In this mode cursor keys can be used to scroll across terminal screen. To exit scrollback mode press **ESC**.

Part III

Applications

Chapter 8

Overview of Internet Transport and Applications

8.1 Introduction

8.1.1 TCP/IP Networking in a Nutshell

In this section, we provide a brief review of the TCP/IP architecture from the point of view of an Operating System. TCP/IP defines the rules that computers must follow to communicate with each other over the Internet. Browsers and Web servers use TCP/IP to communicate with each other, e-mail programs use TCP/IP for sending and receiving e-mails and so on. TCP and IP were developed to interconnect different networks designed by different vendors (Internet) and it was initially successful because it delivered a few basic services that everyone needs (file transfer, electronic mail, remote logon) across a very large number of devices.

Architectures for handling data communication are all composed of layers and protocols. TCP/IP considers two layers: **transport layer** and **network layer**. The transport layer provides an process-to-process service to the application layer, while the network layer provides computer-to-computer services to the transport layer. Finally, the network layer is build on top of a data link layer, which in turn is build over a physical layer. Within each layer, we have several protocols. A protocol is a description of the rules computers must follow to communicate with each other. The architecture is called TCP/IP because TCP and IP are the main protocols for the transport an network layers respectively; but actually, the essential protocols of the TCP/IP architecture in a top-down view are (see also Figure 8.1):

- Transport layer:
  - TCP (Transmission Control Protocol).
  - UDP (User Datagram Protocol).
- Network layer:
  - IP (Internet Protocol).
  - ICMP (Internet Control Message Protocol).

In Unix-like systems, the previous protocols are implemented in the Kernel. In particular, the transport protocols provide the most widely used interface for communication between user processes. In other words, if a process in a user space (application) wants to communicate with another process in another (or the same) user space, it can use the transport protocols with the appropriate system calls and parameters.

- **TCP** provides applications with a full-duplex communication, encapsulating its data over IP datagrams. TCP communication is connection-oriented because there is a handshake of three messages between the kernel TCP instances related to each process before the actual communication between the final processes is possible. These TCP instances may reside on the same, or in different Kernels (computers). The TCP communication is managed as a data flow, in other words, TCP is not message-oriented. TCP adds support to detect errors or lost data and to trigger retransmission until the data is correctly and completely received, so TCP is a reliable protocol.
- **UDP** is the other protocol you can use as transport layer protocol. UDP is similar to TCP, but much simpler and

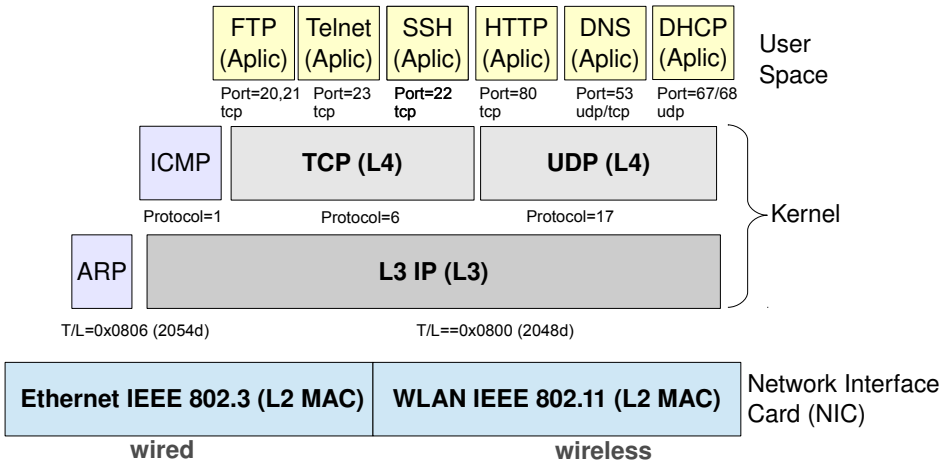


Figure 8.1: Essential TCP/IP protocols and their relationships.

less reliable. For instance, this protocol does not manage packet retransmissions. UDP is message-oriented and each UDP packet or UDP datagram is encapsulated directly within a single IP datagram.

- **IP** is the main protocol of the network layer and it is responsible for moving packets of data between computers also called “hosts“. IP is a connection-less protocol. With IP, messages (or other data) are broken up into small independent "packets" and sent between computers via the Internet. IP is responsible for routing each packet to the correct destination. IP forwards or routes packets from node to node based on a four byte destination address: the IP address. When two hosts are not directly connected to a data link layer network, we need intermediate devices called ”routers“. The IP router is responsible for routing the packet to the correct destination, directly or via another router. The path the packet will follow might be different from other packets of the same communication.
- **ICMP** messages are generated in response to errors in IP datagrams, for diagnostic or routing purposes. ICMP errors are always reported to the original source IP address of the originating datagram. ICMP is connection-less and each ICMP message is encapsulated directly within a single IP datagram. Like UDP, ICMP is unreliable. Many commonly-used network utilities are based on ICMP messages. The most known and useful application is the `ping` utility, which is implemented using the ICMP ”Echo request“ and ”Echo reply“ messages.

Finally, as shown in Figure 8.1, we may have many applications in user space that can use the TCP/IP architecture like telnet, ftp, http/html, ssh, X window, etc. The layers of Figure 8.1 show how application data can be encapsulated to be transmitted over the TCP/IP network. In particular, it is shown that applications can select TCP or UDP as transport layer. Then, these protocols are encapsulated over IP, and IP can select several data link layer technologies like Ethernet, Wireless LAN, etc. to send the data.

8.1.2 Client/Server Model

The model

The client/server model is the most widely used model for communication between processes, generically called ”interprocess communication”. In this model, when there is a communication between two processes, one of them acts as client and the other process acts as server (see Figure 8.2). Clients make requests to a server by sending messages, and servers respond to their clients by acting on each request and returning results. One server can generally support numerous clients.

In Unix-systems, server processes are also called daemons. In general, a daemon is a process that runs in the background, rather than under the direct control of a user. Typically daemons have names that end with the letter ”d“. For example, `telnetd`, `ftpd`, `httpd` or `sshd`. On the contrary, typically, client processes are not daemons. Clients are application processes that usually require interaction with the user. In addition, clients are run in the

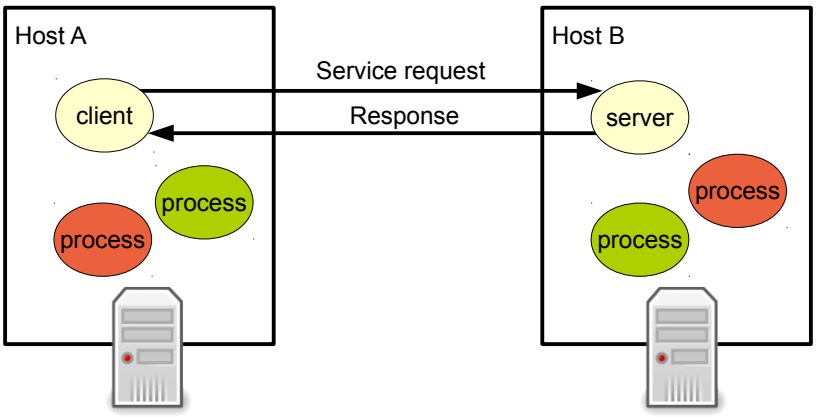


Figure 8.2: Generic client/server model.

system only when needed, while daemons are always running in background so that they can provide their service when requested. **As the client process is who initiates the interprocess communication, it must know the address of server.** One of the ways of implementing this model is to use the system calls of the ”socket interface“. A socket is an endpoint of a bidirectional inter-process communication. The socket interface provides the user space applications with the system calls necessary to enable client/server communications. These system calls are specific to client and to server applications. By now, we will simplify this issue saying that servers open sockets for ”listening“ for client connections and clients open sockets for connecting to servers. In practice, there are several socket implementations for client/server communications using different architectures. Some of the two most popular socket domains are:

- **Unix sockets**. The Unix sockets domain is devoted to communications inside a computer or host. The addresses used in this domain are filenames[1].
- **TCP/IP sockets**. These are the most widely used for client/server communications over networks.

Client/Server TCP/IP

In the TCP/IP domain, the address of a process is composed of: (i) an identifier called **IP address** that allows reaching the destination “user space” or host in which the server process is running, (ii) an identifier of the process called **transport port** and (iii) the **transport protocol** used. So, the client process needs to know these three parameters to establish a TCP/IP socket with a network daemon (server).

To facilitate this aspect, Internet uses a scheme called: well-known services or well-known ports. In this scheme, ports are split basically into two ranges. Port numbers below 1024 are reserved for well-known services and port numbers above 1024 can be used as needed by applications in an ephemeral way. The original intent of service port numbers was that they fall below port number 1024. Of course, the number of services exceeded that number long ago. Now, many network services occupy the port number space greater than 1024[2]. For example, the X server, which is a graphics server listens on port 6000. In this case, we say that this port is the “default” port for this service. As mentioned, the tuple {IP address, protocol, port} serves as ”process identifier” in the TCP/IP network. Some remarkable issues about this tuple are the following:

- **IP address.** Hosts do not have a unique IP address. Typically, they have at least an IP per network interface. Then, we can use in the tuple interchangeably any of the IP addresses of any of the host interfaces to identify a process within a host.
- **Ports and protocol.** Ports are commonly called ”bound“ when they are attached to a given socket (which has been opened by some process). Ports are unique on a host (not interface) for a given transport protocol. Thus,

[1] Further details about this architecture are beyond the scope of this document.
[2] In most Unix-like systems, only the root user can start processes using ports under 1024.

at a certain instant of time, only one process can bind a port for a given protocol. When a server daemon has bound a port, we typically say that it is ”listening“ on that port.

8.1.3 Sockets in Unix

TCP/IP communications where developed in the context of Unix systems, so Linux obviously manages TCP/IP sockets. Inside a Unix-like system the interface “socket” adds an abstraction level in the Kernel. The socket interface was defined and implemented by the BSD (Berkeley Software Distribution) workgroup. If your Kernel implements BSD sockets (most Unix systems do) then your system contains a system call called *socket*. Through this call, you can create TCP or UDP network sockets as client or server providing the tuple {IP address, protocol, port}. When we use the *socket* system call, the Kernel returns a file descriptor that can be used to send and receive data through the network. File descriptors for network communications receive the name of socket descriptors (*sd*) but essentially, they are the same as file descriptors for regular files from the user point of view. Actually, the kernel stores socket descriptors and file descriptors in the same table for a process so the numbers assigned to either *sd* or *fd* must be unique in the system.

Note. Recall that in Unix-like systems, we have also sockets Unix for local inter-process communications. These sockets are similar to TCP/IP ones, but instead of the network and network parameters, they use as server address a “file” name.

In the following sections, we show how to configure the basic parameters of a network interface and we review some of the most popular services that are deployed using TCP/IP networks. The services that we are going to discuss are build over TCP and they use the client/server model.

8.2 Basic Network Configuration

8.2.1 `ifconfig`

The `ifconfig` command is the short for interface configuration. This command is a system administration utility in Unix-like operating systems to configure, control, and query TCP/IP network interface parameters from the command line interface. Common uses for `ifconfig` include setting an interface's IP address and netmask, and disabling or enabling a given interface. In its simplest form, `ifconfig` can be used to set the IP address and mask of an interface. Syntax:

```
# ifconfig IF @IP netmask MASK
```

Example:

```
# ifconfig eth0 192.168.0.1 netmask 255.255.255.0
```

8.2.2 `route`

The `route` command is present at any Unix-like system and it is used to add, remove or view routes. The most commonly used syntax of `route` is the following:

```
# route (add|del) -net @NET netmask MASK [gw @IP dev IF]
```

We can also use the CIDR notation `@NET/X` and we can view the current routing table with the `-n` option, which means view the routing table numerically (without name resolution). We can add routing entries for hosts with the `-host` option. A host route is the most specific entry that can be used since the destination must match completely the 32 bits of the routing entry. Finally, we can define a default route with:

```
# route add default gw @IP
```

Example:

```
alice:~# route add -net 10.0.0.192/26 gw 10.0.0.31
alice:~# route -n
Kernel IP routing table
Destination     Gateway     Genmask         Flags Metric Ref  Use Iface
10.0.0.192      10.0.0.31   255.255.255.192 UG    0      0      0 eth0
10.0.0.0        0.0.0.0     255.255.255.128 U     0      0      0 eth0
```

Note. In Linux, you can configure the host/router behavior in real time and without rebooting the OS by setting a 0 (host) or a 1 (router) in the file: `/proc/sys/net/ipv4/ip_forward`.

8.2.3 Permanent Network Configuration

The configurations made with `route` and `ifconfig` commands are ephemeral. To make the network configuration permanent, in Linux distros like Debian or Ubuntu, the majority of network setup can be done via the interfaces configuration file at /etc/network/interfaces. Here, you can give your network card an IP address (or use dhcp), set up routing information, configure IP masquerading, set default routes and much more. Next, we can see an example:

```
auto eth0
    iface eth0 inet static
        address 192.168.0.10
        netmask 255.255.255.0
        gateway 192.168.0.1
auto eth1
    allow-hotplug eth1
    iface eth1 inet dhcp
nameserver 10.1.1.1
```

The previous file sets the `eth0` interface to the IP address 192.168.0.10/24 and sets the default gateway to 192.168.0.1. Also sets the `eth1` interface to receive the network parameters via DHCP. Finally, sets a DNS server to 10.1.1.1. If you change the configuration of this file, you have to restart "networking" to enable the changes:

```
alice:~# /etc/init.d/networking restart
```

8.2.4 Services

The file /etc/services is used to map port numbers and protocols (TCP/UDP) to service names. Service names can be used by programs. Example:

```
alice:~# less /etc/services
# Network services, Internet style
# Note that it is presently the policy of IANA to assign a single well-known
# port number for both TCP and UDP; hence, officially ports have two entries
# even if the protocol doesn't support UDP operations.
# Updated from http://www.iana.org/assignments/port-numbers and other
# sources like http://www.freebsd.org/cgi/cvsweb.cgi/src/etc/services .
# New ports will be added on request if they have been officially assigned
# by IANA and used in the real-world or are needed by a debian package.
# If you need a huge list of used numbers please install the nmap package.

tcpmux          1/tcp                           # TCP port service multiplexer
echo            7/tcp
echo            7/udp
discard         9/tcp           sink null
discard         9/udp           sink null
systat          11/tcp          users
daytime         13/tcp
daytime         13/udp
...
```

As you can observe, the service "echo" uses the port 7 and the service "daytime" uses the port 13.

8.2.5 `netstat`

The command `netstat` (network statistics) is a tool that displays network connections (both incoming and outgoing), routing tables, and a number of network interface statistics. You should take a look at the `man` of `netstat`. In Table 8.1, we show some of the most significant parameters of this command.

Table 8.1: *`netstat` commonly used parameters.*

-t	TCP connections.
-u	UDP connections.
-l	listening sockets.
-n	addresses and port numbers are expressed numerically and no attempt is made to determine names.
-p	show which processes are using which sockets (you must be root to do this).
-r	contents of the IP routing table.
-i	displays network interfaces and their statistics.
-c	continuous display.
-v	verbose display.
-h	displays help at the command prompt.

For example, to view the current processes listening to TCP ports in the numeric form we can use the following command:

```
alice:~# netstat  -tnlp
Active Internet connections (only servers)
Proto Recv-Q Send-Q   Local Address    Foreign Address  State      PID/Program name
tcp        0      0   0.0.0.0:80       0.0.0.0:*        LISTEN     1690/apache2
tcp        0      0   :::22            :::*             LISTEN     1037/sshd
```

In the output of the previous command we can observe two network daemons: `apache2` with PID 1690 listening to port 80 and `sshd` with PID 1037 listening to port 22.

Another typical example is to view the state of the current TCP connections between clients and servers. For this purpose you can use the following command:

```
alice:~# netstat -tnp
Active Internet connections (w/o servers)
Proto Recv-Q Send-Q Local Address        Foreign Address   State       PID/Program name
tcp        1      0 192.168.0.1:33133     10.0.0.8:80      CLOSE_WAIT  2368/lynx
tcp        0      0 192.168.1.1:45565     10.7.5.8:7       ESTABLISHED 16671/nc
```

8.2.6 `lsof`

The `lsof` command means "list open files" and it is used to report a list of all open files and the processes that opened them. Open files in the system include disk files, pipes and **network sockets**. For example:

```
alice:~# $ lsof -a -p 4578 -d0-10
COMMAND PID    USER   FD   TYPE DEVICE SIZE/OFF NODE NAME
nc     4578   user1    0u   CHR  136,2    0t0    5 /dev/pts/2
nc     4578   user1    1u   CHR  136,2    0t0    5 /dev/pts/2
nc     4578   user1    2u   CHR  136,2    0t0    5 /dev/pts/2
nc     4578   user1    3u  IPv4 149667    0t0  TCP localhost:48911->localhost:12345(ESTABLISHED)
```

The previous command shows us that there is a process called `nc` (netcat, which we will discuss next) with PID 4578 which has four filedescriptors opened. Interestingly, we can observe that the fd=3 is connected to a TCP connection.

8.3 ping

The ping command is used to test the reachability of a host on an IP network. It also measures the round-trip time of messages sent from the originating host to the destination host. The command operates by sending an echo-request ICMP message to the target host and waiting for an echo-replay ICMP message as response. The ping command also measures the time from transmission to reception (round-trip time) and records any packet loss. The results of the test are printed in the form of a statistical summary of the response packets received, including the minimum, maximum, and the mean round-trip times, and sometimes the standard deviation of the mean. The ping command may be run using various options that enable special operational modes such as specifying the ICMP message size, the time stamping options, the time to live parameter, etc. A couple of simple examples are the following:

```
$ ping 192.168.0.2
$ ping -c1 192.168.0.2
```

The first command line sends an echo-request ICMP message each second. You have to type CRL+c to stop it. The second command line sends just a single echo-request and the command ends when the corresponding echo-replay is received or after a timeout.

8.4 netcat

The netcat command is a really useful command for management tasks related to network and it is available in most Unix-like platforms, Microsoft Windows, MAC, etc. It is known as the "Swiss Army Knife of networking" and it is very useful because it can be used to open raw TCP and UDP sockets in client and server modes.

Actually, there are two versions of netcat: one from openbsd and the traditional one. In this document we use the traditional one. To install this version type:

```
# apt-get install netcat-traditional
```

In fact, when you type nc or netcat these are symbolic links. After installing the netcat-traditional package, you have to change these symbolic links to point to the traditional version:

```
# rm /etc/alternatives/nc /etc/alternatives/netcat
# ln -s /bin/nc.traditional /etc/alternatives/nc
# ln -s /bin/nc.traditional /etc/alternatives/netcat
```

Now, in its simplest use, netcat works as a client:

```
$ nc hostname port
```

In this example "hostname" is the name or IP address of the destination host and "port" is the port of the server process. The above command will try to establish a TCP connection to the specified host and port. The command will fail if there is not any server listening on the specified port in the specified host.

We can use the -l (listening) option to make Netcat work as a server:

```
$ nc -l -p port
```

The previous command opens a socket on the specified port in server mode or for listening. In other words, with the -l option the Netcat process is waiting for a client that establishes a connection. Once a client is connected, the behavior of Netcat until it dies (e.g. CRL+c) is as follows (see Figure 8.3):

- The netcat processes read data from stdin (fd=0) and write these data to fd=3 in the client or to fd=4 in the server.
- On the other hand, the data received from the network is read from fd=3 (client) or fd=4 (server) and then, written to fd=1 (stdout).
- Note. To implement netcat we can use the C system calls write() and read(), these system calls use as parameter the fd.

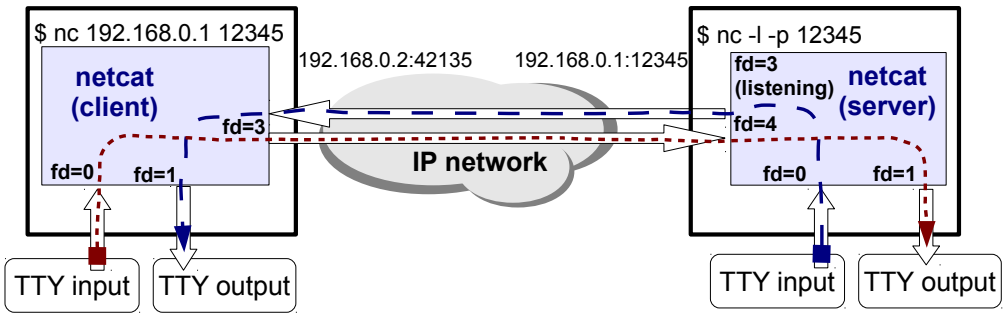


Figure 8.3: File descriptors and `netcat`.

As a first example, we are going to use `netcat` to run a server and a client that establish a TCP socket on the local host. To do this, type:

```
$ nc -l -p 12345
```

The above command creates a socket for TCP listening on port 12345. To view the state of connections we can use the `netstat` command. To verify that there is a TCP socket (option -t) in listening mode (option -l) on port 12345 you can type the following command:

```
$ netstat -tnlp | grep 12345
tcp        0      0 0.0.0.0:12345           0.0.0.0:*               LISTEN    64590/nc
```

Using the following command, we can see that there is no established connection on port 12345:

```
$ netstat -tn | grep 12345
```

Now, we are going to open another pseudo-terminal and run a `nc` in client mode to establish a local TCP connection on port 12345:

```
$ nc localhost 12345
```

An equivalent command to the previous one is `nc 127.0.0.1 12345`. This is because "localhost" is the name for the address 127.0.0.1 which is the default IP address for the *loopback* interface. Now, from a third terminal, we can observe with `netstat` that that the TCP connection is established:

```
$ netstat -tn | grep 12345
tcp        0      0 127.0.0.1:48911         127.0.0.1:12345         ESTABLISHED
tcp        0      0 127.0.0.1:12345         127.0.0.1:48911         ESTABLISHED
```

We can also use lsof to see the file descriptor table of the Netcat processes. The `nc` client process (which in this example has PID=4578) has the following file descriptor table:

```
$ lsof -a -p 4578 -d0-10
COMMAND  PID    USER   FD   TYPE DEVICE SIZE/OFF NODE NAME
nc      4578   user1    0u   CHR  136,2      0t0    5 /dev/pts/2
nc      4578   user1    1u   CHR  136,2      0t0    5 /dev/pts/2
nc      4578   user1    2u   CHR  136,2      0t0    5 /dev/pts/2
nc      4578   user1    3u  IPv4 149667      0t0  TCP localhost:48911->localhost:12345
(ESTABLISHED)
```

The `nc` server process (which in this example has PID=4577) has the following file descriptor table:

```
$ lsof -a -p 4577 -d0,1,2,3,4,5
COMMAND  PID    USER   FD   TYPE DEVICE SIZE/OFF NODE NAME
nc      4577   user1    0u   CHR  136,3      0t0    6 /dev/pts/3
nc      4577   user1    1u   CHR  136,3      0t0    6 /dev/pts/3
```

```
nc      4577   user1    2u   CHR  136,3      0t0    6 /dev/pts/3
nc      4577   user1    3u  IPv4 149636      0t0  TCP *:12345 (LISTEN)
nc      4577   user1    4u  IPv4 149637      0t0  TCP localhost:12345->localhost:48911
(ESTABLISHED)
```

The main options that we will use with `netcat` (traditional version) are shown in Table 12.1.

Table 8.2: *Netcat Options*

-h	show help.
-l	listening or server mode (waiting for incoming client connections).
-p port	local port.
-u	UDP mode.
-e `cmd`	execute `cmd` after the client connects..
-v	verbose debugging (more with -vv).
-q secs	quit after EOF on stdin and delay of secs.
-w secs	timeout for connects and final net reads.

A couple of interesting applications of `netcat`:

- Transfer files:

```
$ cat file.txt | nc -l -p 12345 -q 0
```

- Create a remote terminal:

```
$ nc -l -p 12345 -e /bin/bash
```

We have to point out that the main limitation of `netcat` is that when it runs as server it cannot manage multiple connections. For this purpose, you need to use a specific server for the service you want to deploy or program your own server using a language that let you use system calls (like C).

Finally, we show another example, an "echo server" with `netcat`. The limitation is that this server is able to manage only one client at one time. We can use the following script:

```
1 #!/bin/bash
2 # nc-echo.sh
3 while true
4 do
5 nc -l -p 12345 -e /bin/cat
6 done
```

If you execute the server script and then connect with a client, it is interesting to observe the open files of the `cat` process on the server-side. First, let's see observe the established connections:

```
$ netstat -tnp
Active Internet connections (w/o servers)
Proto Local Address    Foreign Address  State       PID/Program name
tcp   127.0.0.1:41426  127.0.0.1:12345  ESTABLISHED 14688/nc
tcp   127.0.0.1:12345  127.0.0.1:41426  ESTABLISHED 14687/cat
```

As you observe we used the localhost for the TCP connection between the server and the client. Then, you can use `lsof` to list the open files of `cat`:

```
$ lsof -a -p 14687 -d0-10
COMMAND PID   USER FD TYPE DEVICE NAME
cat     14687 user 0u IPv4 39942  TCP localhost:12345->localhost:41426
cat     14687 user 1u IPv4 39942  TCP localhost:12345->localhost:41426
cat     14687 user 2u CHR  136,6  /dev/pts/6
```

As you can observe, the `netcat` server process before ending its execution, starts a `cat` process and connects the standard input and standard output of the `cat` process to the socket established with the client.

8.5 Sockets with Bash

We can open and use client TCP/IP sockets with Bash[3] too. This allows us to create shell-scripts that can act as network clients. For this purpose, Bash handles two special devices that are used as an indication that our intention is to open a TCP/IP network socket instead of a regular file. These files are:

- **/dev/tcp/hostname/port**. When opening this file, actually the Bash attempts to open a TCP socket with the server process identified in the TCP/IP network by the tuple {hostname,TCP,port}.
- **/dev/udp/hostname/port**. When opening this file, actually the Bash attempts to open a UDP socket with the server process identified in the TCP/IP network by the tuple {hostname,UDP,port}.

We must point out that the directories /dev/tcp and /dev/udp do not exist neither are they actual devices. These names are only handled internally by Bash to open TCP/IP network sockets. A constrain is that bash can only open TCP/IP network sockets in client mode. For opening a socket in sever mode (listening), we have to use a external command like `netcat` (which will be discussed in the following section) or create a program with a programming language like C and use the appropriate system calls. Let's illustrate the use of sockets with Bash by some examples. If we type:

```
$ exec 3<>/dev/tcp/192.168.0.1/11300
```

The previous command opens for read and write a client TCP socket connected with a remote server at IP address 192.168.0.1 and port 11300. As you see, we use `exec` in the same way we use it to open file descriptors of regular files. Then, to write to the socket, we do the same as with any other file descriptor:

```
$ echo "Write this to the socket" >&3
```

To read from the socket we can type:

```
$ cat <&3
```

8.6 Commands Summary

Table 8.3 summarizes the commands used within this section.

Table 8.3: Commands related to network applications.

`ifconfig`	configure/view interfaces.
`route`	configure/view routes.
`netstat`	displays system sockets.
`lsof`	lists open files (fd) including socket descriptors.
`ping`	processes echo-request and echo-replay ICMP messages.
`nc or netcat`	opens network sockets as client or server.
`firefox`	A WEB browser.
`lynx`	A console WEB browser.
`apache2`	An HTTP server.

[3] In fact, to have this capability, our Bash must be compiled with the –enable-net-redirections flag.

Chapter 9

Basic Network Applications

9.1 TELecommunication NETwork (TELNET)

9.1.1 What is TELNET?

TELNET or TELecommunication NETwork is a standard Internet protocol for emulating a terminal in a remote host using an IP network. The TELNET service follows the client/server model and uses TCP. TELNET is a well-known service over TCP and its default port is 23 (see Figure 9.1).

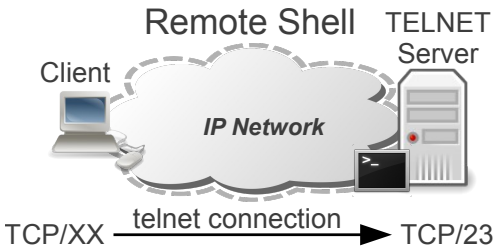


Figure 9.1: Remote Shell Service

9.1.2 Practical TELNET

To use the client, you just need to open a terminal and type `telnet`. After you type the command, you will enter in the telnet "sub-shell", which accepts specific subcommands of `telnet`. The most used and useful subcommand is "`open`", which starts a connection with the specified IP address (or hostname) and with port 23 (this is the default port, but you can specify a different port after the IP address). You can type the subcommand "`help`" to obtain a list o possible subcommands. Next, we show two different ways of connecting with a TELNET server. The first way is to use the `open` subcommand:

```
$ telnet
telnet>open 192.168.0.1
```

A simpler and faster way is to use parameters in the command-line:

```
$ telnet 192.168.0.1 23
```

The port number (23) is optional. You can skip it because this is the default port for the TELNET service. Once the connection is established, TELNET provides a bidirectional interactive text-oriented communication and the commands you type locally, are executed remotely in the host at the other end of the TCP/IP network. You can exit the

remote terminal by typing “exit“. If at any moment you need to return to the `telnet` sub-shell, you can to type “CRL+ALTGR+]“.

9.1.3 TELNET Protocol

Regarding the TELNET protocol, it is worth to mention several issues:

- User data is interspersed in-band with TELNET control information in an 8-bit oriented data connection over the Transmission Control Protocol (TCP).
- All data octets are transmitted over the TCP transport without altering them except the value 255.
- Line endings also may suffer some alterations. The standard says that: ”The sequence CR LF must be treated as a single new line character and used whenever their combined action is intended; the sequence CR NULL must be used where a carriage return alone is actually desired and the CR character must be avoided in other contexts“. So, to achieve a bare carriage return, the CR character (ASCII 13) must be followed by a NULL character (ASCII 0), so if a client finds a CR character alone it will replace this character by CR NULL.

Note. Many times, `telnet` clients are used to establish interactive raw TCP connections. Despite most of the times there will be no problem with this, it must be taken into account that the `telnet` clients apply the previous rules and thus they might alter some little part of the data stream. If you want a pure raw TCP connection, you should use the `netcat` tool. Finally, we must point out that because of security issues with telnet, its use for the purpose of having a remote terminal has waned in favor of SSH (which is discussed later).

9.2 File Transfer Protocol (FTP)

9.2.1 What is a FTP?

File Transfer Protocol (FTP) is a standard Internet protocol for transmitting files between computers (hosts) on the Internet. FTP uses IP and it is based on a client-server model. FTP utilizes separate **control** and **data connections** between the client and server. The default server port for control data is 21 and the default server port for data exchanging is 20 (see Figure 9.2).

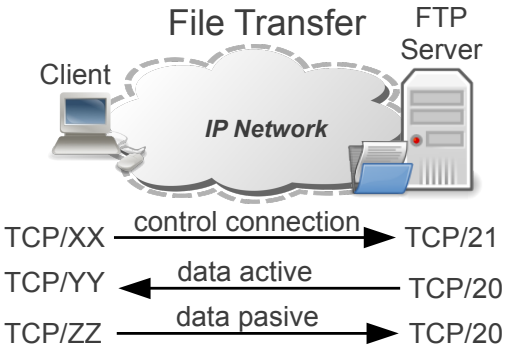


Figure 9.2: File Transfer Service

The control connection remains open for the duration of the session and it is used for session management (commands, parameter negotiation, passwords, etc.) exchanged between the client and server. For example, ”RETR filename“ would transfer the specified file from the server to the client. Due to this two-port structure, FTP is considered an out-of-band, as opposed to an in-band protocol such as TELNET. The server responds on the control connection with three digit status codes in ASCII with an optional text message, for example ”200 OK“ means that the last command was successful. The numbers represent the code number and the optional text represent explanations.

9.2.2 Active and passive modes

FTP can be run in active or passive mode, which determine how the **data connection** is established:

- **In active mode**, the client sends the server the IP address and port number on which the client will listen, and the server initiates the data connection. In situations where the client is behind a firewall or unable to accept incoming TCP connections, passive mode may be used.

- **In passive mode**, the client sends a PASV command to the server and receives an IP address and port number in return. The client uses these to open the data connection to the server.

9.2.3 Data representations

While transferring data over the network, four data representations can be used:

- **ASCII** mode: used for text. Data is converted, if needed, from the sending host's character representation to "8-bit ASCII" before transmission, which uses line endings type CR+LF.
  Note. Some ftp clients change the representation of the received file to the representation of receiving host's line ending. In this case, this mode can cause problems for transferring files that contain data other than plain text.
- **Binary** mode (also called image mode): the sending machine sends each file byte for byte, and the recipient stores the byte-stream as it receives it.
- EBCDIC mode: use for plain text between hosts using the EBCDIC character set. This mode is otherwise like ASCII mode.
- Local mode: Allows two computers with identical setups to send data in a proprietary format without the need to convert it to ASCII.

ASCII and binary are the data representations used in practice.

9.2.4 Data transfer modes

Data transfer can be done in any of three modes:

- **Stream** mode: Data is sent as a continuous stream, relieving FTP from doing any processing. Rather, all processing is left up to TCP.
- Block mode: FTP breaks the data into several blocks (block header, byte count, and data field) and then passes it to TCP.
- Compressed mode: Data is compressed using an algorithm (usually run-length encoding).

In practice, the stream data transfer is the one used.

9.2.5 Practical FTP

Next, we discuss the `ftp` command-line client and `ftpd` server for Linux. The `ftp` client supports active and passive mode, ASCII and binary transmissions and only stream mode. To connect to a `ftpd` server, we can use one of the following options:

```
$ ftp name
$ ftp 192.168.0.1
$ ftp user@192.168.0.1
```

To establish an FTP session you must know the ftp username and password. In case the session is anonymous, typically, you can use the word "anonymous" as both username and password. When you enter your own loginname and password, it returns the prompt "ftp>". This is a sub-shell in which you can type several subcommands. As summary of these subcommands is shown in Table 9.1.

On the other hand, to use the `ftp` client in passive mode (default is active mode) you can type either `ftp -p` or `pftp`. Finally, it is worth to mention that you can also use FTP through a Browser (and also with a several available graphical applications). Browsers such as Firefox allow typing the following in the URL bar:

Table 9.1: Most used *ftp subcommands.*

`open`	opens an FTP session.
`close`	closes and FTP session.
`quit`	exits `ftp`.
`rmdir`	removes a directory in the server.
`mkdir`	creates a directory in the server.
`delete`	removes files in the server.
`get`	downloads a file from the server to the client.
`mget`	downloads multiple files from the server to the client using file expansions (like *).
`put`	uploads a file from the client to the server.
`mput`	uploads multiple files from the client to the server using file expansions (like *).
`type ascii`	selects ascii mode.
`type binary`	selects binary mode.
`!`	executes a command in the local shell.
`cd`	remotely change directory (in the server).
`lcd`	locally change directory (in the client). This is the same as !cd.
`ls`	lists files in the remote directory.
`!ls`	lists files in the local directory.
`pwd`	print remote working directory.
`!pwd`	print local working directory.
`verbose`	show debug info.
`status`	show the configuration parameters.
`help`	help.

```
ftp://ftp.upc.edu
ftp://ftpusername@ftp.upc.edu
ftp://ftpusername:password@ftp.upc.edu
```

9.3 Secure Shell (SSH)

9.3.1 What is SSH?

The SSH protocol is used to obtain an encrypted end-to-end TCP connection between a client (`ssh`) and a server (`sshd`) over a TCP/IP network (see Figure 9.3). The `sshd` daemon listens to port 22 TCP by default. SSH encrypts all traffic (including passwords), which effectively eliminates eavesdropping, connection hijacking, and other attacks.

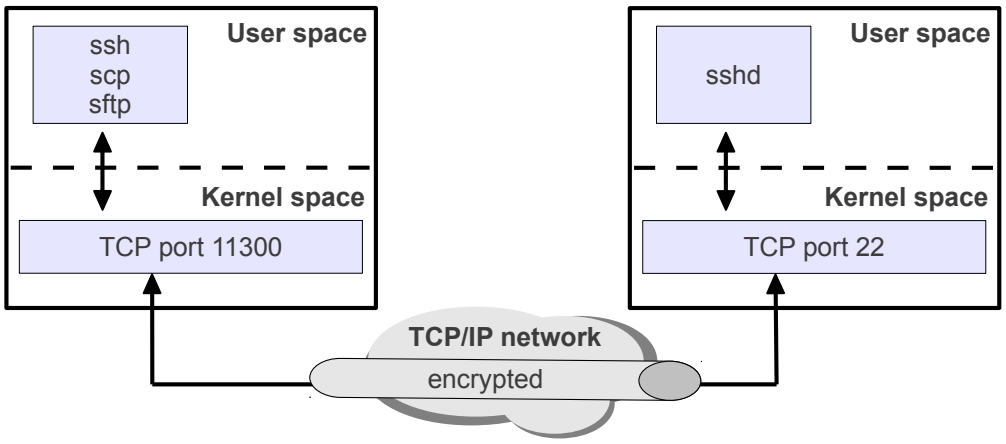


Figure 9.3: SSH client/server example.

9.3.2 Services with SSH

Let's start showing how SSH works with some examples.

Remote Terminal

The first example is to use the `ssh` client to connect to a `sshd` server to obtain a remote terminal. This is the same idea as TELNET, but it is secure. You can achieve this by typing:

```
user1$ ssh 192.168.0.1
```

The previous command will try to establish a SSH session over a TCP socket between the client `ssh` and a `sshd` server listening on 192.168.0.1:22. By default SSH assumes that you want to authenticate with the user you are currently using (environment variable USER). If you want to use a different user for login on the remote host, simply use remoteusername@hostname, such as in this example:

```
user1$ ssh user2@192.168.0.1
```

SSH also offers the file transfer facility between machines on the network and is highly secure, with SSH being an encrypted protocol. Two significant data transfer utilities that use the SSH protocol are SCP and SFTP.

Secure Copy (SCP)

SCP stands for Secure Copy. The command `scp` is essentially a client program that uses the SSH protocol to send and receive files over an encrypted SSH connection. You can transfer files from the client to the server and vice versa. For example, the basic command that copies a file called file.txt **from the client to the server**:

```
$ scp file.txt username@remotehost:
```

The previous command copies the file "file.txt" to the home directory of the user "username" (probably /home-/username) at the server. The lack of a destination filename just preserves the original name of the file. If you want to specify a **new name** for the file on the remote host, simply give the name after the colon:

```
$ scp file.txt username@remotehost:anothername.txt
```

If you want to copy a file to a **directory relative to the home directory** you can type:

```
$ scp file.txt username@remotehost:mydirectory/anothername.txt
```

You can also use **absolute paths** typing "/" after the colon.

```
$ scp file.txt username@remotehost:/tmp
```

To **recursively** copy a directory to the server use the "-r" option. The following command copies a directory named "Documents" to the home directory of the user on the remote host.

```
$ scp -r Documents username@remotehost:
```

To copy **from the server to the client** just reverse the from and to:

```
$ scp username@remotehost:file.txt .
```

Secure File Transfer Protocol (SFTP)

SFTP stands for Secure File Transfer Protocol. It is a secure implementation of the traditional FTP protocol using a SSH session. Let us take a look at how to use the `sftp` command:

```
$ sftp user@hostname
```

For example:

```
user@192.168.0.1:~$ sftp 192.168.0.2
Connecting to 192.168.0.2...
user@192.168.0.2's password:
sftp> cd django
sftp> ls -l
drwxr-xr-x 2 user user 4096 Apr 30 17:33 website
sftp> cd website
sftp> ls
__init__.py __init__.pyc manage.py settings.py settings.pyc
urls.py urls.pyc view.py view.pyc
sftp> get manage.py
Fetching /home/user/django/website/manage.py to manage.py
/home/user/django/website/manage.py 100% 542 0.5KB/s 00:01
sftp>
```

Some of the commands available under `sftp` are:

- `cd`. Changes the current directory on the remote machine.
- `lcd`. Changes the current directory on localhost.
- `ls`. Lists the remote directory contents.
- `lls`. Lists the local directory contents.
- `put`. Send/upload files to the remote machine from the current working directory of the localhost. Supports wildcards for choosing files based on patterns (like *).
- `get`. Receive/download files from the remote machine to the current working directory of the localhost. Supports wildcards for choosing files based on patterns (like *).

9.3.3 Start/Stop `sshd`

The configuration of the `sshd` server is /etc/ssh/sshd_config. If you change the configuration of the daemon you have to stop and start it to apply the changes. As most of the network daemons, `sshd` can be started and stopped under Debian Linux using a script under the directory /etc/init.d. In particular, to stop `sshd` type:

```
# /etc/init.d/ssh stop
```

To start the daemon type:

```
# /etc/init.d/ssh start
```

9.4 Web Server

9.4.1 Apache2

HTTP servers (or WEB servers) listen by default to TCP port 80. The "Apache WEB server", commonly referred to as Apache, is a WEB server software notable for playing a key role in the initial growth of the World Wide Web. In our case, we are going to use its second version: the `apache2` daemon.

9.4.2 Basic Configuration

Ubuntu and other Debian-based distros store the Apache 2.0 configuration files in the directory /etc/apache2. Actually, this configuration file is used to load other configuration files. In Ubuntu, one of these other configuration files is ports.conf, which contains the Listen directives telling `apache2` what IP address and port to listen to.

In the configuration files of the WEB server, we can also find a parameter called "Document Root", which tells us where we can find the WEB resources. Without going into de details, typically, the Document root for `apache2` is /var/www. In this directory we can find a file called index.html that is an HTML file that contains the initial WEB page of the server.

9.4.3 WEB Browsers

To access to the resources of a WEB server you have to use a WEB browser. The WEB browser is the client application of this service. We will use `firefox` (graphical browser) and `lynx` (textual browser). With `firefox` you have to type in the URL bar the address of the server. For example: http://10.1.1.1. Using `lynx` you can type in a terminal the following command to access to a WEB server:

```
$ lynx http://10.1.1.1
```

9.4.4 Start/Stop `apache2`

If you change the configuration of the daemon you have to stop and start it to apply the changes. As most of the network daemons, `apache2` can be started and stopped under Debian Linux using a script under the directory /etc/init.d. In particular, to stop `apache2` type:

```
# /etc/init.d/apache2 stop
```

To start the daemon type:

```
# /etc/init.d/apache2 start
```

9.5 Super Servers

9.5.1 What is a Super Server?

A way of implementing a network service is to use a stand-alone daemon. A stand-alone daemon listens to a port and serves client requests associated to that port. However, for simple services that are used not very often, it is more efficient to use a "super-server" or "super-daemon". **A super-daemon listens to several ports**. A super-daemon uses memory more efficiently, when compared to running each daemon individually in stand-alone when servers have a low burden beacuse the processes of servers are executed only when needed. On the other hand, a dedicated or stand-alone server that intercepts the traffic directly is preferable for services that have frequent traffic.

In Linux, `inetd` is the typical super-daemon. Each time a client requests a service managed by `inetd`, the super-daemon uses its configuration file (/etc/inetd.conf) to determine which server program should manage the request and executes the corresponding server program. In more detail (see Figure 9.4):

- **For each TCP connection `inetd` creates a process and connects STDIN and STDOUT of this process to the socket established with the client**. No network code is required in the server program since `inetd` hooks the socket directly to STDIN and STDOUT of the spawned process.
- UDP sockets are generally handled by a single application-specific server instance that handles all packets on that port.
- Some simple services, such as "daytime", are handled directly by `inetd`, without spawning an external application-specific server.

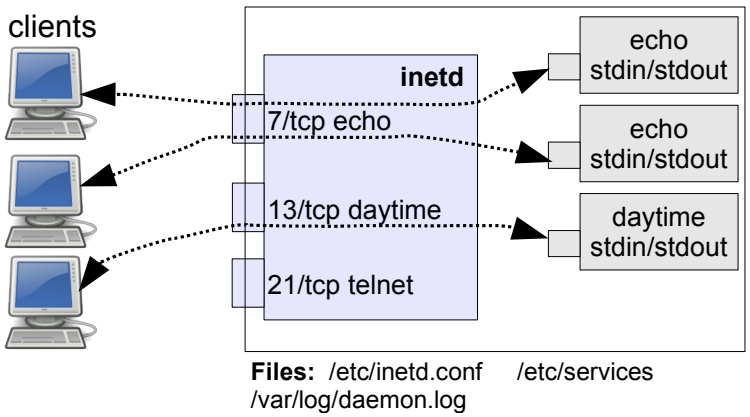


Figure 9.4: How `inetd` works.

We would like to remark that when a daemon is started stand-alone it must open its service port for listening and manage incoming service requests. On the other hand, when a service is managed with `inetd`, the server program can just use STDIN and STDOUT, which are inherited from `inetd`. In this case, as mentioned, the server program does not need any specific code to manage the network.

9.5.2 Configuration

The `inetd` super-daemon is configured using the file /etc/inetd.conf. Each line in /etc/inetd.conf contains the following fields:

```
service_name socket_type protocol {wait|nowait} user server_program [server_program_arguments]
```

For example, the configuration line for the telnet service is something similar to:

```
telnet   stream   tcp     nowait   root      /usr/sbin/in.telnetd
```

The previous configuration-line tells `inetd` to execute the server program "/usr/sbin/in.telnetd" for each incoming telnet connection (port 23 TCP). Notes:

- Lines in /etc/inetd.conf starting with # are comments, i.e. inactive services.
- Be careful to not to leave any space at the beginning of `inetd` configuration lines.

On the other hand, service names and ports are mapped in the configuration file /etc/services. You can check the default port for the telnet service typing:

```
$ cat /etc/services | grep telnet
telnet          23/tcp
...
```

The previous command tells us that the telnet service uses the TCP port number 23. On the other hand, it is also useful to check the log file where `inetd` writes its messages. In a Debian system this file is /var/log/daemon.log.

9.5.3 Start/Stop `inetd`

Each time you change the configuration of `inetd` you have to stop and start it to apply the changes. As most of the network daemons, `inetd` can be started and stopped under Debian Linux using a script under the directory /etc/init.d. In particular, to stop `inetd` type:

```
# /etc/init.d/openbsd-inetd stop
```

To start the super-daemon type:

```
# /etc/init.d/openbsd-inetd start
```

9.5.4 Creating a Service under `inetd`

In short, to create a service under `inetd` you have to:

- Create the server program using STDIN and STDOUT (no network code is required). This program can be a simple script.
- Choose a name and a port for the service.
- Set the port/name of the service in the /etc/services file.
- Configure the service for `inetd` using the file /etc/inetd.conf.
- Stop and start `inetd`.

Following the previous steps, let's configure the service "my-echo" on port 12345. This is going to be an echo service. The first step is to add the configuration line in /etc/inetd.conf:

```
my-echo stream tcp nowait root /root/my-echo.sh
```

Restart `inetd` and observe the log file where `inetd` writes its messages (in a Debian system is /var/log/daemon.log). Taking a look at the log file you will observe it is complaining about the port name:

```
# tail -f /var/log/daemon.log
inetd[1173]: my-echo/tcp: unknown service
```

Let's add the port number for our service "my-echo" in the file /etc/services. The file /etc/services is the system file to map a port number/protocol to a service name. In this example, we will add the following line to /etc/services:

```
my-echo          12345/tcp       # By telematics
```

Restart `inetd` and try to connect to the service. Let's observe the log file:

```
# tail -f /var/log/daemon.log
execv /root/my-echo.sh: No such file or directory
```

Create the file my-echo.sh with the following content:

```
cat
```

The command `cat` without parameters reads writes to STDOUT what it reads from STDIN. Try to connect to the service and observe the log file:

```
# tail -f /var/log/daemon.log
inetd[1204]: execv /root/my-echo.sh: Permission denied
```

Give permissions to the script:

```
# chmod u+x /root/my-echo.sh
```

Try to connect to the service and observe the log file:

```
# tail -f /var/log/daemon.log
inetd[1207]: execv /root/my-echo.sh: Exec format error
```

We need a script "formally correct", to do so, you must include the initial line with the corresponding interpreter (/bin/bash in our example):

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

Finally, let's check the open files of `inetd` and of the processes executed by `inetd`. Open two clients with `netcat` to the port 12345 and observe the related processes:

```
ps ux | grep -E 'my-echo|cat' | grep -v grep
root      1216  0.0  1.9   2640  1164 ?        Ss   00:34   0:00 /bin/bash /root/my-echo.sh
root      1217  0.0  0.6   1640   388 ?        S    00:34   0:00 cat
root      1218  0.0  1.9   2644  1168 ?        Ss   00:35   0:00 /bin/bash /root/my-echo.sh
root      1219  0.0  0.6   1644   392 ?        S    00:35   0:00 cat
```

Now, you can observe that the STDIN, STDOUT and STDERR of these processes are connected to the corresponding socket. For the first connection:

```
virt1:~# lsof -a -p 1216 -d0-10
COMMAND    PID USER   FD   TYPE DEVICE SIZE NODE NAME
my-echo.s 1216 root    0u  IPv4   1681       TCP 10.1.1.1:my-echo->10.1.1.2:3257 (ESTABLISHED)
my-echo.s 1216 root    1u  IPv4   1681       TCP 10.1.1.1:my-echo->10.1.1.2:3257 (ESTABLISHED)
my-echo.s 1216 root    2u  IPv4   1681       TCP 10.1.1.1:my-echo->10.1.1.2:3257 (ESTABLISHED)

virt1:~# lsof -a -p 1217 -d0-10
COMMAND  PID USER   FD   TYPE DEVICE SIZE NODE NAME
cat     1217 root    0u  IPv4   1681       TCP 10.1.1.1:my-echo->10.1.1.2:3257 (ESTABLISHED)
cat     1217 root    1u  IPv4   1681       TCP 10.1.1.1:my-echo->10.1.1.2:3257 (ESTABLISHED)
cat     1217 root    2u  IPv4   1681       TCP 10.1.1.1:my-echo->10.1.1.2:3257 (ESTABLISHED)
```

For the second connection:

```
virt1:~# lsof -a -p 1218 -d0-10
COMMAND    PID USER   FD   TYPE DEVICE SIZE NODE NAME
my-echo.s 1218 root    0u  IPv4   1686       TCP 10.1.1.1:my-echo->10.1.1.2:3258 (ESTABLISHED)
my-echo.s 1218 root    1u  IPv4   1686       TCP 10.1.1.1:my-echo->10.1.1.2:3258 (ESTABLISHED)
my-echo.s 1218 root    2u  IPv4   1686       TCP 10.1.1.1:my-echo->10.1.1.2:3258 (ESTABLISHED)

virt1:~# lsof -a -p 1219 -d0-10
COMMAND  PID USER   FD   TYPE DEVICE SIZE NODE NAME
cat     1219 root    0u  IPv4   1686       TCP 10.1.1.1:my-echo->10.1.1.2:3258 (ESTABLISHED)
cat     1219 root    1u  IPv4   1686       TCP 10.1.1.1:my-echo->10.1.1.2:3258 (ESTABLISHED)
cat     1219 root    2u  IPv4   1686       TCP 10.1.1.1:my-echo->10.1.1.2:3258 (ESTABLISHED)
```

Finally, we check the open files of `inetd`. First let's figure out the PID of inetd:

```
virt1:~# ps ux | grep inetd | grep -v grep
root      1188  0.0  1.0   1816   612 ?        Ss   00:06   0:00 /usr/sbin/inetd
```

Then, check its open files:

```
virt1:~# lsof -a -p 1188 -d0-10
COMMAND  PID USER   FD   TYPE DEVICE SIZE  NODE NAME
inetd   1188 root    0u   CHR    1,3      65538 /dev/null
inetd   1188 root    1u   CHR    1,3      65538 /dev/null
inetd   1188 root    2u   CHR    1,3      65538 /dev/null
inetd   1188 root    4u  IPv4   1618        TCP *:my-echo (LISTEN)
inetd   1188 root    5u  IPv4   1621        TCP *:ftp (LISTEN)
inetd   1188 root    6u  IPv4   1624        TCP *:auth (LISTEN)
```

9.6 Commands summary

Table 9.2 summarizes the commands used within this section.

9.7 Practices

For the following exercises we are going to use the scenario *netapps-basic*. This scenario has two virtual machines **virt1** and **virt2**, connected with a virtual switch. The **phyhost** is also connected to the virtual switch with its `tap0`

Table 9.2: Commands related to basic network applications.

`telnet`	TELNET remote terminal client.
`telnetd`	TELNET server.
`ftp`	FTP client.
`ftpd`	FTP server.
`ssh`	SSH remote terminal client.
`scp`	Secure copy client.
`sftp`	Secure FTP client.
`sshd`	SSH server.
`apache2`	A WEB server.
`inetd`	A Super server.

interface. Each virtual machine has two consoles enabled (0 and 1). To start this scenario type the following command on your **phyhost**:

```
phyhost$ simctl netapps-basic start
```

Once the simulation has been started, you can get the console 0 of **virt1** and so on typing:

```
phyhost$ simctl netapps-basic get virt1 0
```

When you finish the practices, do not forget to stop the simulation:

```
phyhost$ simctl netapps-basic stop
```

Exercise 9.1– In this exercise you must answer some basic questions about networking configuration.

1. Open all the consoles: *virt1.0* (console 0 in **virt1**), *virt1.1*, *virt2.0* and *virt2.1*. Then, figure out which is the port number of the service *daytime*.

   Tip. Take a look at the file `/etc/services`.

2. In a windows-like OS, which port number you expect for the *daytime* service?

3. Annotate the MAC and IP addresses of each interface on **virt1** and **virt2**. Which command have you used?

4. Assign the IP addresses 192.168.0.1 and 192.168.0.2 to the Ethernet interfaces of **virt1** to **virt2**, respectively.

5. Send 3 ICMP echo-requests from **virt2** to **virt1** with the `ping` command.

6. Find information and explain what is and for what can be used the loopback (`lo`) interface. Why `lo` does not have a MAC address?

7. Configure the original IP addresses on each virtual machine.

Exercise 9.2– In this exercise you will start, stop and configure some network daemons (services).

1. Using the `netstat` command, list the TCP services that are active in **virt1** under `inetd`. Describe the service names and ports used and check that the results are consistent with the configuration of `inetd` (/etc/inetd.conf).

2. In **virt1**, edit the configuration of `inetd` and activate the service *daytime*. Restart the super-daemon and check that the port of *daytime* is being listened by `inetd`. Use `nc` to connect to this service from **virt2** and describe what you observe.

3. In **virt1**, check that the SSH daemon is listening to TCP port 22. Stop the SSH daemon and check that now the TCP port 22 is not being listened.

4. In **virt1**, change the configuration of the SSH daemon to listen to port 2222 instead of 22. Check your configuration. To finish this exercise, in **virt1** change the configuration again of the SSH daemon to listen to its default port.

Exercise 9.3– In this exercise, you have to use `netcat` to create your own stand-alone servers.

1. Start a new capture on `tap0` with `wireshark` in the **phyhost** and try the following command:

```
virt1.0$ nc -l -p 12345
```

   Describe what the previous command does. It is a client or a server? Describe how can you know the ports and open files related to this `netcat` process. Now try:

```
virt2.0$ nc 10.1.1.1 12345
```

   Describe the ports and the open files used by each `netcat` process. Send some text from each machine and terminate the connection. Describe the network parameters of the captured traffic. Use also the follow TCP stream option and describe how it works.

2. Start a new capture on `tap0` with `wireshark` in the **phyhost**. In **virt1**, create a server with `netcat` that listens on port 23456 and transfers the file /etc/services.

   Tip. Use the option -q 0 to quit after the transmission of the file.

   Connect to the previous server with a `nc` from **virt2** and store the file sent by the server with the name "file.txt". For captured traffic, describe what you observe using the option follow TCP stream.

3. Start a new capture on `tap0` with `wireshark` in the **phyhost**. Repeat the steps of the previous exercise but this time using UDP.

   Note. In the client you have to type two times "ENTER" to start the data reception.

   For captured traffic, describe what you observe and the differences between the previous capture. Notice that in this case you can use the option follow UDP stream.

4. In the **phyhost**, create a server with `netcat` that listens on port 12345 and emulates the daytime service.

   Tip. Use the `date` command.

   After several seconds, try the service from **phyhost** with `nc`. Which interface you should use to capture the traffic of the previous network exchange?

   Which is the actual date (minutes and seconds) that you observe as output in the client `nc`? It is correct? Which do you think that is the cause of this behavior?

5. In **virt1**, create a server with `netcat` that listens on port 22333 and provides to the client the amount of free disk in the machine.

   Tip. Use the `df -h` command.

   Try the service from **virt2** with `nc`.

Exercise 9.4– In this exercise, you must create a service under `inetd`.

1. In **virt1**, implement a service using `inetd` (without using a `netcat` as server) that listens on port 22333 and that provides the amount of free disk in the system to the client. Explain the configuration that you have to make. In this case, can you connect to this service several times? To this respect, explain how this implementation works and the differences with respect to using a simple server with `netcat`.

   Tip. If you have problems, check /var/log/daemon.log which is the `inetd`'s log file.

Exercise 9.5– In this exercise, we are going to analyze the Web service.

1. In **virt1**, start the `apache2` WEB server. To do so, type the following command (notice that we avoid viewing STDERR):

   ```
   virt1.0# /etc/init.d/apache2 start 2> /dev/null
   ```

   Do yo see a configuration line in `inetd` for `apache2` why? Find out the PID of the process that is listening on port 80 in **virt1**. Is this process `inetd`? Describe how you do it.

2. In **virt1**, edit the file /var/www/index.html with `vi` and without modifying the HTML tags, change some of the text of *It works!*. Using a console in **virt2** and the `lynx` WEB browser, display the web page of **virt1**.

3. Start a new capture on `tap0` with `wireshark` in the **phyhost**. Also in the **phyhost**, give the IP address 10.1.1.3/24 to the `tap0` interface. Open a `firefox` in the **phyhost** and use it to see the web page served by **virt1**. For captured traffic, use the follow TCP stream option of `wireshark` and roughly comment the protocols that you see.

Exercise 9.6– In this exercise, we are going to analyze the remote terminal service which TELNET and SSH.

1. Start a new capture on `tap0` with `wireshark` in the **phyhost**. Try to establish a remote terminal on **virt1** using `telnet` from **virt2**. Did it work? Explain the output of the `telnet` command and the captured traffic. Which do you think that is cause of the behavior observed.

2. Activate the TELNET service under `inetd` in **virt1**.

   Note. Be careful to not to leave any space at the beginning of `inetd` configuration lines.

   Check your configuration with `netstat` and start a capture on `tap0` with `wireshark` in the **phyhost**. Try to establish a remote terminal from **virt2** to **virt1** using `telnet`. Did it work? explain what you observe. Take a look at the /var/log/daemon.log and describe the messages in that file.

3. In general, the root user cannot access with TELNET to a remote machine. To enable to this possibility, you have to enable one or more TTYs in the file /etc/securetty. Each line with pts/X enables a TTY or possible TELNET connection. Start a new capture on `tap0` with `wireshark` in the **phyhost**, enable a TTY for the root user in **virt1** (uncommenting one this lines at the beginning of /etc/securetty) and try again to establish a remote terminal from **virt2** to **virt1** using `telnet` from *virt2.0*. Using **virt2.1** and the TELNET session in *virt2.0*, type a `netstat` command with the appropriate parameters to check the ports used by the TELNET connection and the processes that have registered these ports. Explain the differences of what you see in *virt2.0* and *virt2.1*. Check the file descriptors used by the telnet client and the telnet server.

4. Under the TELNET session, create an empty file called file.txt in the home directory of the root user in **virt1** and exit. With *virt1.0* check the creation of the file.

5. Use the follow TCP stream option of `wireshark` and comment the TELNET protocol. What do you think about the security of this protocol?

6. Capture in `tap0` and try an SSH session from **virt2** to **virt1**. Use the follow TCP stream option of `wireshark` and comment the differences between SSH with TELNET about ports used and security.

Exercise 9.7– In this exercise, we are going to analyze the file transfer service with FTP, SCP and SFTP.

1. Start a new capture on `tap0` with `wireshark` in the **phyhost**. Then, establish an FTP session from **virt2** to **virt1** using the root user and the console *virt2.0*. Did it work? Use the follow TCP stream option of `wireshark` and comment what you observe.

2. The FTP access for the root user is blocked by default as with TELNET. To enable it, you have to modify the configuration file **/etc/ftpusers** by removing or commenting (using the # symbol at the beginning of the line) the line for the root user. Using the console *virt1.0* allow the FTP access for the root user.

3. Start a new capture on `tap0` with `wireshark` in the **phyhost** and establish an FTP session from **virt2** to **virt1** using the console *virt2.0*. Using the console *virt2.1* check the ports and the file descriptors used in **virt2**. Using the console *virt1.0* check the ports and the file descriptors used in **virt1**.

4. Use the FTP session to get all the files in /usr/bin that start with "z" and exit. Which is the default data representation for these transmissions?

5. Use the follow TCP stream option of `wireshark` to comment the FTP dialogue previously generated. Figure out also how the data (files) are transferred and which ports are used.

6. Look at the files you have downloaded in **virt2**. Check the permissions. Are these permissions the same as in the server? When you finish, remove these files in the client and exit the FTP session.

7. Start a new capture on `tap0` with `wireshark` in the **phyhost** and establish an SFTP session from **virt2** to **virt1** using the console *virt2.0*. Use the SFTP session to get all the files in /usr/bin that start with "z". Use the follow TCP stream option of `wireshark` and roughly comment the SFTP dialogue.

8. Start a new capture on `tap0` with `wireshark` in the **phyhost**. Also in the **phyhost**, give the IP address 10.1.1.3/24 to the `tap0` interface and create a file called "file.txt" with the content "hello world". Using `scp` and the root user, transfer this file to the directory /root of **virt1**. Use the follow TCP stream option of `wireshark` and roughly comment the SCP dialogue.

Chapter 10

Unix GUI

10.1 Linux/Unix GUI

10.1.1 Introduction

Unix has been a multiuser, multitasking, timesharing operating system since its beginnings. When the time came to develop a Graphical User Interface (GUI) system that could run primarily under Unix, these concepts were kept in mind and incorporated into the design. As shown in Figure 10.1, one of the guiding philosophies is that the GUI is achieved through the co-operation of separate components, rather than everything being entwined in one huge mass. The advantage of this is that a particular part of the system can be changed simply by replacing the relevant component. This is why the GUI in Unix has a pretty complex design, which has often been mentioned as a disadvantage. However, because of its design, it is also really versatile. As shown in Figure 10.1, the components of the GUI run all in user space. Also the client/server architecture is used between many of these components.

A good reference for further reading and also the inspiration of this text is [**?**].

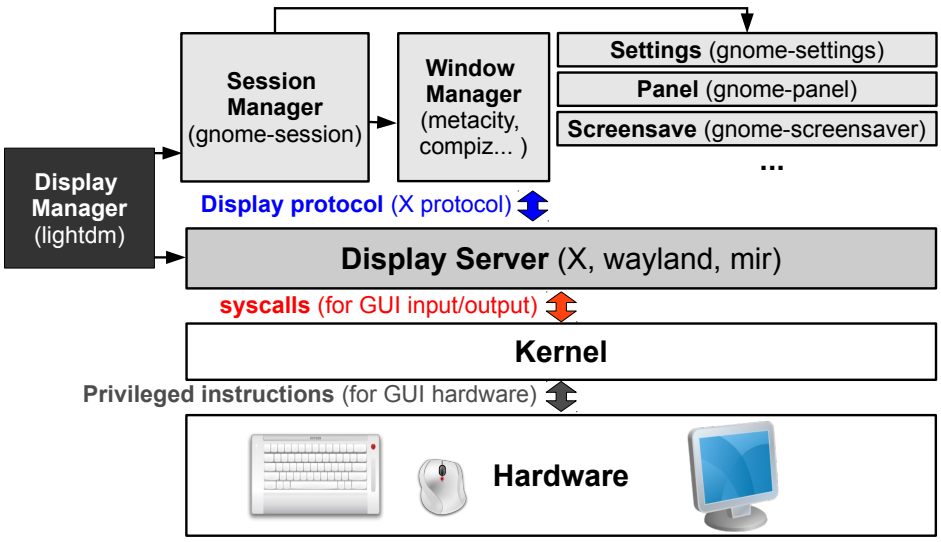


Figure 10.1: GUI Organization.

10.1.2 Display Server

The core component of the GUI is the **display server**. A display server is a program whose primary tasks are:

- Coordinate the input and output of its clients to and from the hardware related to GUI.
- Coordinate the communication between its clients.

The display server maintains exclusive control of the display and services requests from the clients. This includes reading the mouse and keyboard (using the proper system calls) and rely all this information to the client or clients involved in these graphical actions. These clients of course, will have to react accordingly. Applications (clients) only need to know how to communicate with the server, and need not be concerned with the details of talking to the actual graphics display device. At the most basic level, a client tells the server stuff like "draw a line from here to here", or "render this string of text, using this font, at this position on-screen". The display server is built as a user space application and it provides for its clients an abstraction layer for the display on top of the kernel.

This would be no different from just using a graphics library to write our application. However the Unix GUI model goes a step further. It does not constrain the client being in the same computer as the server. The display server communicates with its clients using a display protocol, which is a communications protocol that in many implementations can be used over a network. This is called a "network transparent protocol".

The most widely known display server implementation is the X[1] server and its associated X protocol, which is network transparent. X was designed as a client-server architecture. The applications themselves are the clients; they communicate with the server and issue requests, also receiving information from the server. X provides a library called Xlib, which handles all low-level client-server communication tasks. Then, the client has to invoke functions contained within Xlib to get work done.

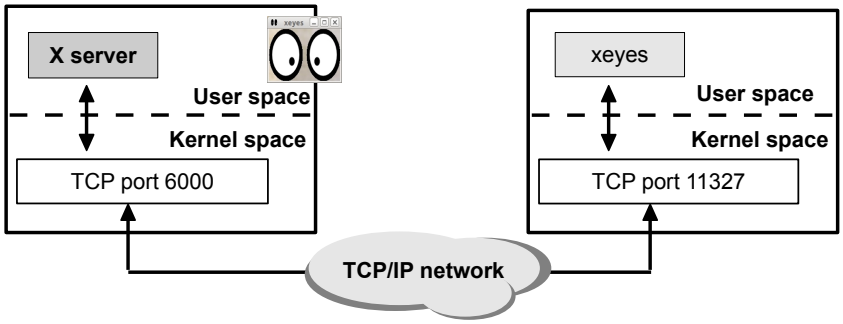


Figure 10.2: X client/server example.

As shown in Figure 10.2, the X server listens by default on port TCP 6000. The user's computer running the server and the remote applications being the clients, often confuses novice people, because the terms seem reversed. But X takes the perspective of the application, rather than that of the end-user: the X server provides display and I/O services to applications, so it is a server; applications use these services, thus they are clients.

10.1.3 Window Manager

At this point everything seems to be working fine. We have a server in charge of visual output and data input, client applications, and a way for them to communicate between each other. However, being a client, I would like not having to be concerned about what other clients have displayed on the screen. For example, to avoid my graphic objects to be overlapped with the objects of another client. This is smartly solved using rectangle areas are called windows. Then, the client does not care about where it has been displayed on the screen, but it just uses a window and then calls the functions to perform the graphic actions (inside the window) like "draw a line from here to there" or "tell me whether the user is moving the mouse in my screen area".

[1]The X.Org Foundation leads the X project, with the current reference implementation, X.Org Server, available as free and open source software under the MIT License and similar permissive licenses. X is also known as X11 because 11 is the current major version. Other examples of display servers are Wayland and Mir from the Ubuntu team.

In general, we can move and arrange windows, change their size and maximize or minimize them. Virtually everything which appears on the screen is in a window. However, the display server does not do the job of managing windows, it is another program's responsibility to manage the on-screen space. This program is the "window manager". The window manager decides where to place windows, gives mechanisms for users to control the windows' appearance, position and size, and usually provides "decorations" like window titles, frames and buttons, that give us control over the windows themselves. Also it is common that window managers provide keyboard shortcuts to perform their operations.

The window manager is a special client of the display server whose most basic mission is to manage other clients. There are many window managers which support different ways for the user to interact with windows and different styles of window layout, decoration, and keyboard and colormap focus. Most window managers provide additional facilities. For example, many of them provide some way of launching applications like a command box where you can type commands or a menu of some sort. There are also specific client applications whose sole mission is to launch other client applications. In practice, you can find many "program launching" applications you can use on your system. These window managers also differ on the resource consumption they require. Examples are metacity, compiz, kwin, icewm, fvwm, etc.

10.1.4 Desktop System

Libraries provided by display servers are very low level libraries. For example, Xlib (for X) is pretty spartan, and doing things like putting buttons on screen, text, or nice controls like scrollbars, menus, etc. is terribly complicated. Luckily, someone else went to the trouble of programming these controls and giving them to us in a usable form; a library. These controls are usually known as "widgets" and the library is a "widget library".

Since the widget library is the one actually drawing the elements on-screen, as well as interpreting user's actions into input, the library used is largely responsible for each client's aspect and behavior. Just as it happens with window managers, there are many widget libraries or toolkits, the most widely known being Gtk and Qt. Due to the fact that clients can be programmed using several possible different toolkits and we can use different window managers here is where the "mess" begins:

- Each window manager has a different approach to managing the clients; the behavior and decorations are different from one to the next.

- As defined by which toolkit each client uses, applications can also look and behave differently from each other.

- Using lots of different toolkits also increases resource usage. Modern operating systems support the concept of dynamic shared libraries. This means that if I have two or three applications using Gtk, and I have a dynamic shared version of Gtk, then those two or three applications share the same copy of Gtk, both on the disk and in memory. This saves resources. On the other hand, if I have a Gtk application and a Qt application I am now loading two different libraries in memory, one for each of the different toolkits (but the toolkits provide basically the same functionality).

Finally, there are features that one expects from a GUI environment that our current GUI components do not cover. For example, a control panel for configuring the system, a screen-saver, etc. Of course, there are many open source implementations that can provide these features but a desktop environment aims to provide a complete interface to the operating system, and supply its own range of integrated utilities and applications. This convenience and ease of use makes desktops particularly attractive to new users. Under Linux, the two most popular desktop environments are KDE and GNOME:

- KDE uses the Qt toolkit and kwin as window manager.

- GNOME has always tried to be window manager-agnostic. Today it uses mainly compiz and metacity. In addition, GNOME uses the Gtk toolkit, and provides a set of higher-level functions and has its own set of programming guidelines in order to guarantee a consistent behavior between compliant applications.

10.1.5 Session Manager

A session is the collection of applications, settings, and resources (like open files) present on the user's desktop. A session manager is responsible for:

- Starting sessions on a desktop environment. In other words, the session manager starts all the initial processes required to have a session on the desktop. In GNOME this includes the gnome-panel, the gnome-screensaver, etc. In general, in modern Linux desktops you can also configure the additional applications that you want to be executed when your session starts.

- Providing users a possibility to save and restore their sessions. Every application that is session management aware connects to the session manager. A session manager sends commands to his clients telling them to save their state or to terminate. A client must provide the session manager with all information, that is needed to restart the client in the same state, as it is running now. The session manager's task is to take care of this information and to use it when restarting a session. In order to distinguish all clients, the session manager assigns them a unique identifier: the so called client id.

In the X system, a subprotocol named X Session Management Protocol (XSMP) specifies how applications and session managers interact. Of particular importance is that the window manager is able to communicate with the session manager, as the window manager is responsible for the placement of windows and the existence of icons. Applications that cannot store their state can be included in a session, but they do not preserve their state across sessions. The X Window System includes a default session manager called xsm. Other session managers have been developed for specific desktop systems: for example, ksmserver is the default session manager of KDE and gnome-session is the default session manager for GNOME (gnome-session uses D-Bus instead of XSMP).

10.1.6 Display Manager

The last important component of the Unix/Linux GUI is the display manager[2] or login manager. The display manager starts the display server before presenting the user the login screen, optionally repeating when the user logs out. The login screen prompts for a username and password. A session starts when the user successfully enters a valid combination of username and password.

The display manager and the display server can run on different systems. In this case, the display server runs on the computer in front of the user and it is configured to be connected to the corresponding display manager that is running on another computer. In this case, the X Display Manager Control Protocol (XDMCP) is used between the display server and the display manager. XDMCP uses the client/server model with the well-known port 177/UDP for the server. As shown in Figure 10.3, the display server is the XDMCP client and the display manager is the XDMCP server. The display server requests the display manager to start a session in the remote desktop and after that, we can execute GUI applications in this session.

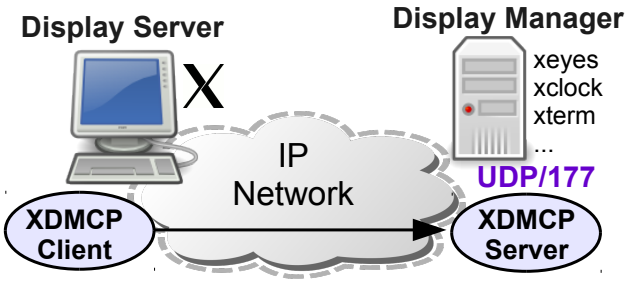


Figure 10.3: XDMCP client/server.

[2]This is the same functionality of getty and login on character-mode terminals.

The X Window System supplies XDM as its standard display manager but there are other display managers. For example, GDM (part of GNOME), KDM (part of KDE) and LightDM (by Canonical Ltd, the creators of Ubuntu). On many Linux distributions, the default display manager is selected in the file /etc/X11/default-display-manager.

10.2 X in Practice

10.2.1 Transport Architecture

For the client/server X communications we can different transport architectures, including Unix sockets and TCP/IP sockets. Which mechanism is used, is determined by the "`displayname`" contained in the DISPLAY environment variable. `displaynames` are in the form: `hostname:displaynumber.screennumber`. Where:

- `hostname` is the name or IP address of the host in which is running the X server.
- `displaynumber` identifies the X server within the host since there can be different X servers running at a time in the same host.
- `screennumber` specifies the screen to be used on that server. Multiple screens can be controlled by a single X server.

When DISPLAY does not contain a hostname, e.g. it is set to :0.0, Unix sockets will be used. When it does contain a hostname, e.g. it is set to localhost:0.0, the X client application will try to connect to the server via TCP/IP sockets (localhost in this example). For example, if you type:

```
$ export DISPLAY="192.168.0.1:0.0"
$ xeyes &
```

The first command-line sets and exports the variable DISPLAY and the second command-line executes an `xeyes`. In this case, we are specifying that the X server in which the `xeyes` are going to be displayed is running in a host that can be reached with IP address 192.168.0.1 and that our target X server is 0 (screen 0). More precisely, when we say server 0, we mean that we expect the X server to be listening to port 6000, which is the default port for server 0. If you want to connect to server 1, you should type:

```
$ export DISPLAY="192.168.0.1:1.0"
```

And this would mean that for X clients launched after setting this display expect an X server listening on port 6001 of host 192.168.0.1. Another way to do remote display is by a command line option that all X applications have: "-display [displayname]". Then, to produce the same result as before but without using the DISPLAY variable, we can type:

```
$ xeyes -display 192.168.0.1:0.0
```

10.2.2 Activate TCP/IP

In many modern Linux distributions, the X server is not configured to use the TCP/IP transport. If we want to use it, we have to enable this feature in the display manager. In new Ubuntu distributions (>= 11.10) the default display manager is LightDM. To enable TCP on the X server just open or create the file /etc/lightdm/lightdm.conf (you must be root) and add the following content:

```
[Seat:0]
xserver-allow-tcp=true
```

Finally, reboot lightdm (or reboot the system):

```
# /etc/init.d/lightdm restart
or
# service lightdm restart
```

10.2.3 X authentication

Finally, there is a small amount of security that the server has to check when it gets a connection from a client. X has an authentication mechanism based on a list of allowed client hosts. The `xhost` command adds (`xhost +hostname`) or deletes (`xhost -hostname`) host names on the list of machines from which the X Server accepts connections. Entering the `xhost` command with no variables shows the current host names with access your X Server and a message indicating whether or not access is enabled. Finally, you can also use `host +` and `host -` to disable or enable respectively the access control list (this is summarized in Table 10.1).

Table 10.1: `xhost` Parameters.

`xhost`	shows the X server access control list.
`xhost +hostname`	Adds hostname to X server access control list.
`xhost -hostname`	Removes hostname from X server access control list.
`xhost +`	Turns off access control (all remote hosts will have access to X server).
`xhost -`	Turns access control back on.

For example, to enable access to our X server from host 192.168.0.1, we should type:

```
$ xhost +192.168.0.1
```

We must notice that for security reasons, executing `xhost` with the options that affect access control is only allowed on the host in which the display server is running.

Part IV

Linux Advanced

Chapter 11

Shell Scripts

11.1 Introduction

A shell is not just an interface for executing commands or applications, it is much more than this. Shells provide us with a powerful programming language for creating new commands called ”shellscripts“ or simply “scripts“. A script serves as “ glue” for making work together several commands or applications. Most Unix commands follow philosophy: “*Keep It Simple Stupid !*”, which means that commands should be as simple as possible and address specific tasks, and then, use the power of scripting to perform more complex tasks. Shellscripts were introduced briefly in Section 2.6. This section, by means of examples, makes a more detailed introduction to the possibilities that scripts offer.

11.2 Quoting

One of the main uses of quoting in when defining variables. Let’s define a variable called MYVAR[1] whose value is the word ”Hello“. Example:

```
$ MYVAR=Hello
```

Notice that there are not any spaces between the equal sign ”=“ and the value of the variable ”Hello”. Now, if we want to define a variable with a value that contains spaces or tabs we need to use quotes. Example:

```
$ MYVAR='Hello World'
```

Single quotes mean that the complete string ’Hello World‘ (including spaces) must be assigned to the variable. Then, a “Hello World” script could be:

```
1  #!/bin/bash
2  MYVAR='Hello world'
3  echo $MYVAR
```

The dollar sign “$” before a variable name means that we want to use its value. Now, we want to use the value of MYVAR to define another variable, say MYVAR2. For this purpose, let’s try the following script:

```
1  #!/bin/bash
2  MYVAR='Hello world'
3  MYVAR2='$MYVAR, How are you?'
4  echo $MYVAR2
```

If you execute the previous script, you will obtain as output: `$MYVAR, How are you?`, which is not the expected result. To be able to use the value of a variable and also quoting (values with spaces) you have to use double

[1] In general, you can use many combination of letters, numbers and other signs to define variables but by convention, we will not use lowercase.

Table 11.1: Types of quotes

	Quotes	Meaning
'	simple	The text between single quotes is treated as a literal (without modifications). In bash, we say that it is not *expanded*.
"	double	The text between double quotes is treated as a literal except for what follows to characters \, `and $.
`	reversed	The text between reverse quotes is interpreted as a command, which is executed and whose output is used as value. In bash, this is known as *command expansion*.

quotes. In general, there are three types of quotes as shown in Table 11.1. Now, if you try the following script you will obtain the desired result:

```
1  #!/bin/bash
2  MYVAR='Hello world'
3  echo "$MYVAR, how are you?"
```

Finally, the third type of quoting are reverse quotes. These quotes cause the quoted text to be interpreted as a shell command. Then, the command is expanded to its output. Example:

```
$ echo "Today is `date`"
Today is Tue Aug 24 18:48:08 CEST 2008
```

On the other hand, we must point out that when you expand a variable, you may have problems if it is immediately followed by text. In this case, you have to use braces {}. Let's see an example:

```
$ MYVAR='Hello world'
$ echo "foo$MYVARbar"
foo
```

We were expecting the output "fooHello worldbar", but we did not obtain this result. This is because the variable expansion of bash was confused. Bash could not tell if we wanted to expand $M, $MY $MYVAR, $MYVARbar, etc. We have to use braces to resolve the ambiguity:

```
$ echo foo${MYBAR}bar
fooHello worldbar
```

Although $MYVAR is faster to type, it is safer to use always ${MYBAR}.

11.3 Positional and special parameters

Often, you will need that your script can process arguments given to it when invoked. These arguments are called positional parameters. These positional parameters are named as: $1 to $N. When N consists of more than a single digit, it must be enclosed in braces like ${N}. The positional parameter $0 is the basename of the script as it was called. For example, create the following script:

```
1 #!/bin/bash
2 echo "$1"
```

Now, execute the script as follows and observe the outputs:

```
$ ./my_script.sh Hello word
Hello
$ ./my_script.sh 'Hello word'
Hello word
```

As shown in the example above, positional parameters are considered to be separated by spaces. You have to use single or double quotes if you want to specify a single positional parameter assigned to a text string that contains multiple words (i.e. a text containing spaces or tabs). For example, create the following script:

```
#!/bin/bash
echo Script name is "$0"
echo First positional parameter $1
echo Second positional parameter $2
echo Third positional parameter  $3
echo The number of positional parameters is $#
echo The PID of the script is $$
```

Now, execute the script as follows and observe the outputs:

```
$ ./my_script.sh first 'second twice' third fourth fifth
Script name is ./my_script.sh
First positional parameter first
Second positional parameter second twice
Third positional parameter third
The number of positional parameters is 5
The PID of the script is 4307
```

As you can observe, there are parameters that have a special meaning: $# expands to the number of positional parameters, $$ expands to the PID of the bash that is executing the script, and $@ expands to all the positional parameters (also $* does this).

In more detail, $@ and $* both expand to all the positional parameters. There are no differences between $* and $@, but there is a difference between "$@" and "$*". "$*" means always one single argument, and "$@" contains as many arguments. Example:

```
$ cat script1.sh
mkdir "$*"
$ cat script2.sh
mkdir "$@"
$ ./script1 dir1 dir2 ; ./script2 dir3 dir4
$ ls | grep dir
dir1 dir2
dir2
dir3
```

"$@" is a special token which means "wrap each individual argument in quotes".

11.4 Expansions

Before executing your commands, bash checks whether there are any syntax elements in the command line that should be interpreted rather than taken literally. After splitting the command line into tokens (words), bash scans for these special elements and interprets them, resulting in a changed command line: the elements are said to be expanded to or substituted to new text and maybe new tokens (words). We have several types of expansions. In processing order they are:

- **Brace Expansion**: create multiple text combinations.
  Syntax: {X,Y,Z} {X..Y} {X..Y..Z}
- **Tilde Expansion**: expand useful pathnames home dir, working dir and previous working dir.
  Syntax: ~ ~+ ~-
- **Parameter Expansion**: how bash expands variables to their values.
  Syntax: $PARAM ${PARAM...}
- **Command Substitution**: using the output of a command as an argument.
  Syntax: $(COMMAND) `COMMAND`
- **Arithmetic Expansion**: how to use arithmetics.
  Syntax: $((EXPRESSION)) $[EXPRESSION]
- **Process Substitution**: a way to write and read to and from a command.
  Syntax: <(COMMAND) >(COMMAND)

- **Filename Expansion**: a shorthand for specifying filenames matching patterns.
  Syntax: *.txt page_1?.html

11.4.1 Brace Expansion

Brace expansions are used to generate all possible combinations with the optional surrounding preambles and postscripts. The general syntax is: `[preamble]{X,Y[,...]}[postscript]` Examples:

```
$ echo a{b,c,d}e
abe ace ade
$ echo "a"{b,c,d}"e"
abe ace ade
$ echo "a{b,c,d}e"              # No expansion because of the double quotes
a{b,c,d}e
$ mkdir $HOME/{bin,lib,doc}     # Create $HOME/bin, $HOME/lib and $HOME/doc
```

The are also a couple of alternative syntaxes for brace expansions using two dots:

```
$ echo {5..12}
5 6 7 8 9 10 11 12
$ echo {c..k}
c d e f g h i j k
$ echo {1..10..2}
1 3 5 7 9
```

Another example with multiple braces:

```
$ echo {a..g..2}{1..10}
a1 a2 a3 a4 a5 a6 a7 a8 a9 a10 c1 c2 c3 c4 c5 c6 c7 c8 c9 c10
e1 e2 e3 e4 e5 e6 e7 e8 e9 e10 g1 g2 g3 g4 g5 g6 g7 g8 g9 g10
```

11.4.2 Tilde Expansion

The tilde expansion is used to expand three specific pathnames:

- Home directory: ~
- Current working directory: ~+
- Previous working directory: ~−

Examples:

```
$ cd /
/$ cd /usr
/usr$ cd ~-
/$ cd ~
~$ cd /etc
/etc$ echo ~+
/etc
```

11.4.3 Parameter Expansion

Parameter expansion allows us to get the value of a parameter (variable). On expansion time, you can do extra processing with the parameter or its value. We have already used the basic parameter expansion when we used $VAR or ${VAR}. Next, we describe two examples of parameter expansion with extra processing:

${VAR:-string} If the parameter VAR is not assigned, the expansion results in 'string'. Otherwise, the expansion returns the value of VAR. For example:

```
$ VAR1='Variable VAR1 defined'
$ echo ${VAR1:-Variable not defined}
Variable VAR1 defined
$ echo ${VAR2:-Variable not defined}
Variable not defined
```

You can also use positional parameters. Example:

```
USER=${1:-joe}
```

${VAR:=string} If the parameter VAR is not assigned, VAR is assigned to 'string' and the expansion returns the assigned value. Note. Positional and special parameters cannot be assigned this way.

More parameter expansions are described in Section 11.9.5.

11.4.4 Command Substitution

Command substitution yields as result the standard output of the command executed. It has two possible syntaxes: `$(COMMAND)` or `` `COMMAND` `` Examples:

```
$ MYVAR=`dirname /usr/local/share/doc/foo/foo.txt`
$ echo $MYVAR
/usr/local/share/doc/foo
$ echo $(basename /usr/local/share/doc/foo/foo.txt)
foo.txt
```

We also can use command substitution together with parameter expansion. Example:

```
$ echo VAR=${VAR1:-Parameter not defined in `date`}
Parameter not defined in Thu Mar 10 15:17:10 CET 2011
```

11.4.5 Arithmetic Expansion

Despite bash is primary designed to manipulate text strings, it can also perform arithmetic calculations. For this purpose, we have arithmetic expansion. The syntax is: ((...)) or $[...]. Let's see some examples.

```
VAR=55              # Asign the value 55 to the variable VAR.
((VAR = VAR + 1))   # Adds one to the variable VAR (notice that we don't use $).
((++VAR))           # C style to add one to VAR (preincrease).
((VAR++))           # C style to add one to VAR (postincrease).
echo $[VAR * 22]    # Multiply VAR by 22
echo $((VAR * 22)) # Multiply VAR by 22
```

We also can use the extern command `expr` to do arithmetic operations. However, if you use `expr`, your script will create a new process, making the processing of arithmetics less efficient. Example:

```
$ X=`expr 3 \* 2 + 7`
$ echo $X
13
```

Note. Bash cannot handle floating point calculations, and it lacks operators for certain important mathematical functions. For this purpose you can use `bc`.

11.4.6 Process Substitution

Process substitution is explained in Section 4.5.

11.4.7 Filename expansion

In file expansion, a pattern is replaced by a list names of files sorted alphabetically. The possible patterns for files expansion are the following:

Table 11.2: Patterns for file expansion

Format	Meaning
*	Any string of characters, including a null (empty) string.
?	Any unique character.
$[List]$	A unique character from the list. We can include range of characters separated by hyphens (-). If the first character of the list is ^ or !, this means any single character that it is not in the list.

Next we show some examples.

```
$ ls *.txt
file.txt  doc.txt  tutorial.txt
$ ls [ft]*
file.txt  tutorial.txt
$ ls [a-h]*.txt
file.txt  doc.txt
$ ls *.??
script.sh file.id
```

More specifically, filename expansions use glob patterns, which are a type of regular expression. Regular expressions are described next.

11.5 Regular Expressions

11.5.1 Introduction

A regular expression is a special text string for describing a search pattern mainly for use in find and replace like operations. In Linux systems, we have to types of regular expressions: **glob patterns** and **regular expressions (regex)**. The glob patterns are the oldest and simplest type of regular expressions used in Unix-like systems. They are used by commands related with the file system like `ls`, `rm`, `cp` etc. On the other hand, regex are more versatile regular expressions and they are used by commands related to string manipulation and filtering like `grep` or `sed` (for a `sed` introduction see Section 11.9.6).

11.5.2 Glob Patterns

We have two sets of globs, basic and extended. The main syntax used by both sets is described next.

Basic globs

The basic globs and their meaning are listed below:

Character	**Meaning**
?	Expands one character.
*	Expands zero or more characters (any character).
`[ ]`	Expands one of the characters inside [ ].

Extended globs

To use the following expansions you have to have activated extended globbing. You can check this with:

```
$ shopt -s extglob
```

The extended globs and their meaning are listed below:

Character	Meaning
?(...|...)	Expands **zero or one** occurrence of the items listed between the pipes
*(...|...)	Expands **zero or more** occurrence of the items listed between the pipes
+(...|...)	Expands **one or more** occurrence of the items listed between the pipes
@(...|...)	Expands **any** occurrence of the items listed between the pipes
!(...|...)	Expands anything **except** the items listed between the pipes

Examples:

```
$ touch 03.txt ; touch {aa,b,c}{00,01,02}.txt

$ ls
03.txt a00.txt  a01.txt  a02.txt  aa00.txt  aa01.txt  aa02.txt
b00.txt  b01.txt  b02.txt  c00.txt  c01.txt  c02.txt

$ ls *(a|b)??.txt
03.txt a00.txt  a01.txt  a02.txt  aa00.txt  aa01.txt  aa02.txt  b00.txt  b01.txt  b02.txt

$ ls ?(a)0*
03.txt  a00.txt  a01.txt  a02.txt

$ ls +(a|?0)*
a00.txt  a02.txt   aa01.txt  b00.txt  b02.txt  c01.txt
a01.txt  aa00.txt  aa02.txt  b01.txt  c00.txt  c02.txt

$ ls @(aa|*2)*
a02.txt  aa00.txt  aa01.txt  aa02.txt  b02.txt  c02.txt

$ ls !(a0*|aa0*)
03.txt b00.txt  b01.txt  b02.txt  c00.txt  c01.txt  c02.txt
```

11.5.3 Regular Expressions (regex)

Regular Expresions provide three main features:

- Concatenation. RE ab matches an input string of ab.
- Union. RE alb matches an input string of a or an input string of b. It does not match ab.
- Closure. RE a* matches the **empty string**, or an input string of a, or an input string of aa, etc.

We have two sets of regex, basic and extended. The main syntax used by both sets is described next.

Basic regex

The basic regex and their meaning are listed below:

\is used to escape special characters.

. A dot character matches any single character of the input line.

^ This character does not match any character but represents the beginning of the input line. For example, `^A` is a regular expression matching the letter A at the beginning of a line.

$ This represents the end of the input line.

`[ ]` A bracket expression. Matches a single character that is contained within the brackets. For example, [abc] matches a, b, or c. [a-z] specifies a range which matches any lowercase letter from a to z. These forms can be mixed: [abcx-z] matches a, b, c, x, y, or z, as does [a-cx-z].

`[^ ]` Matches a single character that is not contained within the brackets.

RE* A regular expression followed by * matches a string of zero or more strings that would match the RE. For example, A* matches A, AA, AAA, and so on. **It also matches the null string (zero occurrences of A)**.

`\( \)` are used for grouping. The RE inside the escaped parenthesis creates a group that can be referenced with `\1` through `\9` (up to nine groups can be created).

`\{ \}` means the same as {m,n} (without backslash) does in Extended regex (see next Section).

Extended regex

The extended regex and their meaning are listed below:

`RE+` A regular expression followed by + matches a string of one or more strings that would match the RE.

`RE?` A regular expression followed by ? matches a string of zero or one occurrences of strings that would match the RE.

`()` which are parenthesis without backslash, are used for grouping in Extended regex.

`{}` is used to create unions. Examples:

- `a{3}` is equivalent to regular expression aaa (exactly `a` three times).
- `a{3,}` is equivalent to `aaaa*` (3 or more `a`).
- `a{,3}` is equivalent to `|a|aa|aaa` (matches the empty string or `a` or `aa` or `aaa`).
- `a{3,5}` is equivalent to regular expression `aaa|aaaa|aaaaa`.

Examples

Next, we provide some examples of regex (basic and extended):

`abc` matches any line of text containing the three letters abc in that order.

`.at` matches any three-character string ending with at, including hat, cat, and bat.

`[hc]at` matches hat and cat.

`[^b]at` matches all strings matched by .at except bat.

`^[hc]at` matches hat and cat, but only at the beginning of the string or line.

`[hc]at$` matches hat and cat, but only at the end of the string or line.

`\[.\]` matches any single character surrounded by [], for example: [a] and [b].
Remark: brackets must be escaped with `\`.

`^.$` matches any line containing exactly one character (the newline is not counted).

`.* [a-z]+ .*` matches any line containing a word, consisting of lowercase alphabetic characters, delimited by at least one space on each side.

Example in a pipeline:

```
$ ps -u $USER | grep '^ [0-9][0-9][0-9]9'
```

The previous command-line shows the processes of the user that have a PID of four digits and that end with the digit 9. More examples of regex are given when `sed` is described in Section 11.9.6.

11.6 Conditional statements

In this section we present conditional statements available for shell scripts.

11.6.1 If

The clause `if` is the most basic form of conditional. The syntax is:

`if` **expression** `then` **statement1** `else` **statement2** `fi`.

Where **statement1** is only executed if **expression** evaluates to true and **statement2** is only executed if **expression** evaluates to false. Examples:

```
if [ -e /etc/file.txt ]
  then
  echo "/etc/file.txt exists"
  else
  echo "/etc/file.txt does not exist"
fi
```

In this case, the expression uses the option `-e file`, which evaluates to true only if "file" exists. Be careful, you must leave spaces between "[" and "]" and the expression inside. With the symbol "`!`" you can do inverse logic. Example:

```
if [ ! -e /etc/file.txt ]
  then
  echo "/etc/file.txt does not exist"
  else
  echo "/etc/file.txt exists"
fi
```

We can also create expressions that always evaluate to true or false. Examples:

```
if true
  then
  echo "this will always be printed"
  else
  echo "this will never be printed"
fi
if false
  then
  echo "this will never be printed"
  else
  echo "this will always be printed"
fi
```

Other operators for expressions are the following.

File Operators:

```
[ -e filename ] true if filename exists
[ -d filename ] true if filename is a directory
[ -f filename ] true if filename is a regular file
[ -L filename ] true if filename is a symbolic link
[ -r filename ] true if filename is readable
[ -w filename ] true if filename is writable
[ -x filename ] true if filename is executable
[ filename1 -nt filename2 ] true if filename1 is more recent than filename2
[ filename1 -ot filename2 ] true if filename1 is older than filename2
```

String comparison:

```
[ -z string ] true if string has zero length
[ -n string ] true if string has nonzero length
[ string1 = string2 ] true if string1 equals string2
[ string1 != string2 ] true if string1 does not equal string2
```

Arithmetic operators:

```
[ X -eq Y ] true if X equals Y
[ X -ne Y ] true if X is not equal to Y
[ X -lt Y ] true if X is less than Y
[ X -le Y ] true if X is less or equal than Y
[ X -gt Y ] true if X is greater than Y
[ X -ge Y ] true if X is greater or equal than Y
```

The syntax ((...)) for arithmetic expansion can also be used in conditional expressions. The syntax ((...)) supports the following relational operators: $==, !=, >, <, >=, y <=$. Example:

```
if ((VAR == Y * 3 + X * 2))
then
   echo "The variable VAR  is equal to Y * 3 + X * 2"
fi
```

We can also use conditional expressions with the OR or with the AND of two conditions:

```
[ -e filename -o -d filename ]
[ -e filename -a -d filename ]
```

Finally, it is worth to mention that **in general it is a good practice to quote variables** inside your conditional expressions. If you don't quote your variables you might have problems with spaces and tabs. Example:

```
if [ $VAR = 'foo bar oni' ]
then
echo "match"
else
echo "not match"
fi
```

In the previous example the expression might not behave as you expect. If the value of `VAR` is "foo", we will see the output "not match", but if the value of `VAR` is "foo bar oni", bash will report an error saying "*too many arguments*". The problem is that spaces present in the value of VAR confused bash. In this case, the correct comparison is: `if [ "$VAR" = 'foo bar oni' ]`. Recall that you have to use double quotes to use the value of a variable (i.e. to allow parameter expansion).

Finally, it is worth to mention that **we can use glob patterns and regex in conditional statements** using the syntax `[[ ... ]]`. When comparing with glob patters we have to use == and when comparing with regex we have to use =~. Examples:

```
[[ $a == z* ]]   # True if $a starts with an "z" (pattern matching).
[[ $a == "z*" ]] # True if $a is equal to z* (literal matching).
```

In the previous example, note that quotes deactivate the meaning of the glob pattern (also happens with regex). Another example with globbing:

```
if [[ $VAR == ??grid* ]] ; then echo $VAR; fi
```

The same example using a regex:

```
if [[ $VAR =~ ^..grid.*$ ]] ; then echo $VAR; fi
```

11.6.2 Conditions Based on the Execution of a Command

Each command has a return value or exit status. Exit status is used to check the result of the execution of the command. If the exit status is zero, this means that the command was successfully executed. If the command failed, the exit status will be non-zero (see Table 11.3).

Table 11.3: Standard exit status

Exit Value	Exit Status
0 (Zero)	Success
Non-zero	Failure
2	Incorrect usage
127	Command Not found
126	Not an executable
128 + N	The command is terminated by signal N

The special variable $? is a shell built-in variable that contains the exit status of the last executed command. If we are executing a script, $? returns the exit status of the last executed command in the script or the number after the keyword `exit`. Next, we show an example:

```
command
if [ "$?" -ne 0]
 then
 echo "the previous command failed"
 exit 1
fi
```

The clause `exit` allows us to specify the exit status of the script. In the previous example we reported failure because one is greater than zero. We can also replace the previous code by:

```
$ command || echo "the previous command failed"; exit 1
```

Finally, we can also use the return code of conditions. Conditions have a return code of 0 is the condition is true or 1 if the condition is false. Using this, we can get rid of the keyword `if` in some conditionals. Example:

```
$ [ -e filename ] && echo "filename exists" || echo "filename does not exist"
```

The first part of the previous command-line evaluates the condition and returns 0 if the condition is true or 1 if the condition is false. Based on this return code, the echo is executed.

11.6.3 for

A `for` loop is a bash programming language statement which allows code to be repeatedly executed. A `for` loop is classified as an iteration statement i.e. it is the repetition of a process within a bash script. The syntax is:

```
for VARIABLE in item1 item2 item3 item4 item5 ... itemK
do
        command1
        command2
        ...
        commandN
done
```

The previous `for` loop executes K times a set of N commands. You can also use the values (item1, item2, etc.) that takes your control VARIABLE in the execution of the block of commands. Example:

```
#!/bin/bash
for X in one two three four
do
echo number $X
done

Script's output:
number one
number two
number three
number four
```

In fact, the loop accepts any list after the key word "*in*", including listings of the file system. Example:

```
#!/bin/bash
for FILE in /etc/r*
do
if [ -d "$FILE" ]
then
echo "$FILE (dir)"
else
echo "$FILE"
fi
done

Script's output:
/etc/rc.d (dir)
/etc/resolv.conf
/etc/rpc
```

Furthermore, we can use *filename expansion* to create the list of files/folders Example:

```
for FILE in /etc/r??? /var/lo* /home/student/* /tmp/${MYPATH}/*
do
cp $FILE /mnt/mydir
done
```

We can use relatives paths too. Example:

```
for FILE in ../* documents/*
do
echo $FILE is a silly file
done
```

In the previous example the expansion is relative to the script location. We can make loops with the positional parameters of the script as follows:

```
#!/bin/bash
for THING in "$@"
do
echo You have typed: ${THING}.
done
```

With the command `seq` or with the syntax (()) we can generate C-styled loops:

```
    #!/bin/bash
    for i in `seq 1 10`;
    do
    echo $i
    done
```

```
    #!/bin/bash
    for (( i=1; i < 10; i++));
    do
    echo $i
    done
```

Remark. Bash scripts are not compiled but interpreted by bash. This impacts their performance, which is poorer than compiled programs. If you need an intensive usage of loops, you should consider using a compiled program (for example, written in C).

11.6.4 while

A `while` loop is another bash programming statement which allows code to be repeatedly executed. Loops with `while` execute a code block while a expression is true. For example:

```
1
2 #!/bin/bash
3 X=0
4 while [ $X -le 20 ]
5 do
6         echo $X
7         X=$((X+1))
8 done
```

The loop is executed while the variable `X` is less or equal (-le) than 20. We can also create infinite loops, example:

```
1
2 #!/bin/bash
3 while true
4  do
5   sleep 5
6   echo "Hello I waked up"
7  done
```

11.6.5 case

A `case` construction is a bash programming language statement which is used to test a variable against set of patterns. Often `case` statement let's you express a series of if-then-else statements that check single variable for various conditions or ranges in a more concise way. The generic syntax of `case` is the following:

```
1 case VARIABLE in
2     pattern1)
3         1st block of code ;;
4     pattern2)
5         2nd block of code ;;
6     ...
7 esac
```

A pattern can actually be formed of several subpatterns separated by pipe character "|". If the VARIABLE matches one of the patterns (or subpatterns), its corresponding code block is executed. The patterns are checked in order until a match is found; if none is found, nothing happens.

For example:

```
1 #!/bin/bash
2 for FILE in $*; do
3   case $FILE in
4    *.jpg | *.jpeg | *.JPG | *.JPEG)
5    echo The file $FILE seems a JPG file.
6    ;;
7   *.avi | *.AVI)
8    echo "The filename $FILE has an AVI extension"
9    ;;
10  -h)
11   echo "Use as: $0 [list of filenames]"
12   echo "Type $0 -h for help" ;;
13  *)
14   echo "Using the extension, I don't now which type of file is $FILE."
15   echo "Use as: $0 [list of filenames]"
16   echo "Type $0 -h for help" ;;
17  esac
18 done
```

The final pattern is *, which is a catchall for whatever didn't match the other cases.

11.7 Formatting output

We have already seen the `echo` command for generating text output. We have also another (more complete) command for that: `printf`. Let us start with a simple example:

```
$ printf "hello printf"
hello printf$
```

As you see there is a different behavior in comparison to `echo` command. No new line had been printed out as it it in case of when using default setting of `echo` command. To print a new line we need to supply `printf` with format string with escape sequence `\n` (new line):

```
$ printf "Hello, $USER.\n\n"
```

or

```
$ printf "%s\n" "hello printf"
hello printf
```

The format string is applied to each argument:

```
$ printf "%s\n" "hello printf" "in" "bash script"
hello printf
in
bash script
```

As you could observe in the previous examples we have used %s as a format specifier. Specifier %s means to print all argument in literal form. The specifiers are replaced by their corresponding arguments. Example:

```
$ printf "%s\t%s\n" "1" "2 3" "4" "5"
1       2 3
4       5
```

The %b specifier is essentially the same as %s but it allows us to interpret escape sequences with an argument. Example:

```
$ printf "%s\n" "1" "2" "\n3"
1
2
\n3
$ printf "%b\n" "1" "2" "\n3"
1
2

3
$
```

To print integers, we can use the %d specifier:

```
$ printf "%d\n" 255 0xff 0377 3.5
255
255
255
bash: printf: 3.5: invalid number
3
```

As you can see %d specifiers refuses to print anything than integers. To `printf` floating point numbers use %f:

```
$ printf "%f\n" 255 0xff 0377 3.5
255.000000
255.000000
377.000000
3.500000
```

The default behavior of %f `printf` specifier is to print floating point numbers with 6 decimal places. To limit a decimal places to 1 we can specify a precision in a following manner:

```
$ printf "%.1f\n" 255 0xff 0377 3.5
255.0
255.0
377.0
3.5
```

Formatting to three places with preceding with 0:

```
$ for i in $( seq 1 10 ); do printf "%03d\t" "$i"; done
001     002     003     004     005     006     007     008     009     010
```

You can also print ASCII characters using their hex or octal notation:

```
$ printf "\x41\n"
A
$ printf "\101\n"
A
```

11.8 Functions and variables

11.8.1 Functions

As with most programming languages, you can define functions in bash. The syntax is:

```
function function_name {
commands...
}
```

or

```
function_name () {
commands...
}
```

Functions can accept arguments in a similar way as the script receives arguments from the command-line. Let us show an example:

```
#!/bin/bash
# file: myscript.sh
zip_contents() {
 echo "Contents of $1: "
  unzip -l $1
}
zip_contents $1
```

In the previous script, we defined the function `zip_contents()`. This function accepts one argument: the filename of a *zip* file and shows its content. Observing the code, you can see that in the script we use the first positional parameter received in the command-line as the first argument for the function. To test the script, try:

```
$ zip -r etc.zip /etc
$ myscript.sh home.zip
```

The other special variables have also a meaning related to the function. For example, after the execution of a function, the variable $? returns the exit status of the last command executed in the function or the value after the keyword `return` or `exit`. The difference between `return` and `exit` is that the former does not finish the execution of the script, while the latter exits the script.

A function may be compacted into a single line, for example to be executed in the command line. For example:

```
$ zip_contents () { echo "Contents of $1: "; unzip -l $1; }
```

A semicolon must follow each command (also the final command).

11.8.2 Variables

Unlike many other programming languages, by default bash does not segregate its variables by type. Essentially, bash variables are character strings, but, depending on context, bash permits arithmetic operations and comparisons on variables. The determining factor is whether the value of a variable contains only digits or not. This design is very flexible for scripting, but it can also cause difficult to debug errors. Anyway, you can explicitly define a variable as numerical using the bash built-in `declare`:

```
declare -i VAR
```

The built-in `declare` can be used also to declare a variable as read-only (`declare -r VAR`).

On the other hand, we will use the convention of defining the name of the variables in **capital letters** to distinguish them easily from commands and functions. Finally, it is important to know that variables have one of the following scopes: **shell** variables, **local** variables, **environment** variables, **position** variables and **special** variables.

Shell Variables

A shell variable, as its name states, is a variable defined for a shell (e.g. a bash). In general, when you define a variable in a script, the variable is defined for the shell that is executing the script. To execute scripts, **the bash in which the script is launched clones itself** and the variables defined in the script are valid **only in this cloned shell**.

On the other hand, you can also define a shell variable in the shell that is attached to a terminal:

```
$ VAR=hello
$ echo $VAR
```

However, the previous variable **will not available to child programs** like a cloned bash that executes a script. If you need that a certain variable **is available for a child program**, you have to make it an **environment variable** as described later.

Local Variables

Simply stated, local variables are those variables used **inside a function**, but there are subtleties about local variables that you must be aware of. In most complied languages like C, when you create a variable inside a function, the variable is placed in a separate namespace regarding the general program.

Let's us consider that you write a program in C in which you define a function called `my_function`, you define a variable inside this function called '`X`', and finally, you define another variable outside the function also called '`X`'. In this case, these two variables have different local scopes so if you modify the variable '`X`' inside the function, the variable '`X`' defined outside the function is not altered. While this is the default behavior of most compiled languages, **it is not valid for bash scripts**.

In a bash script, when you create a variable inside a function, it is added to the script's namespace. This means that if we set a variable inside a function with the same name as a variable defined outside the function, we will override the value of the variable. Furthermore, a variable defined inside a function will exist after the execution of the function. Let's illustrate this issue with an example:

```
#!/bin/bash
VAR="Outside the function"
my_function(){
VAR="Inside the function"
}
my_function
echo $VAR
```

When you run this script the output is ”Inside the function“. If you really want to declare VAR as a local variable, whose scope is only its related function, you have to use the keyword `local`, which defines a local namespace for the function. The following example illustrates this issue:

```
#!/bin/bash
VAR="Outside the function"

my_function(){
local VAR="Inside the function"
}

my_function
echo $VAR
```

In this case, the output is ”Outside the function“, so the shell variable VAR is not affected by the execution of the function.

Environment Variables

As previously mentioned, shell variables are not available for child processes like the bash that executes a script. Let's define a shell variable in a terminal:

```
$ VAR=hello
$ echo $VAR
```

Let's create a script that tries to use the value of VAR like:

```
#!/bin/bash
echo "The content of VAR is $VAR"
```

The output of the previous script is only "The content of VAR is" because VAR is not defined inside the cloned bash that executes the script. Child processes only inherit the **“exported context”**. Variables (and functions) are not exported by default. To export variables and functions and view the exported and current environment you can use the following commands:

- **Export a variable**. Syntax: `export VAR` or `declare -x VAR`.
- **Export a function**. Syntax: `export -f function_name`.
- **View current context**. Syntax: `declare`.
- **View exported context**. Syntax: `export` or `declare -x`.

For example, let's use our script to test environment variables:

```
#!/bin/bash
echo $VAR
```

Then, execute the following:

```
$ VAR=hello ; echo $VAR
hello
$ declare | grep VAR
VAR=hello
$ ./script.sh
$ export VAR ; export | grep VAR
MY_VAR=hello
$ ./script.sh
hello
```

Notice that `script.sh` can use the variable 'VAR' only after it is exported. As mentioned, exported variables and functions get passed on to child processes, but not-exported variables do not. Anyway, if VAR is defined inside the script, the value of the shell variable is the one used in that script. In other words, definitions of shell variables prevail over definitions of environment variables.

On the other hand, you can delete (unset) an environment variable (exported or not) using the command `unset`. Example:

```
$ unset MY_VAR
$ ./env-script.sh
```

When a bash is executed, **a set of environment variables is loaded**. There are several configuration files in which the system administrator or the user can set the initial environment variables for bash. The two main files are:

- ~/.bashrc (per user configuration).
- /etc/profile (system-wide configuration).

To finish this discussion, we would like just to mention that some typical environment variables widely used in Unix-like systems are shown in Table 11.4. Recall that the values of the environment variables `HOME` and `PWD` can also be obtained using tilde expansions: ~ and ~+.

Table 11.4: Typical environment variables

Variable	Meaning
SHELL=/bin/bash	The shell that you are using.
HOME=/home/student	Your home directory.
LOGNAME=student	Your login name.
OSTYPE=linux-gnu	Your operating system.
PATH=/usr/bin:/sbin:/bin	Directories where bash will try to locate executables.
PWD=/home/student/documents	Your current directory.
USER=student	Your current username.
PPID=45678	Your parent PID.

`source` Command

As mentioned, when you execute a script, bash creates a child bash to execute the commands of the script. This is the default behavior because executing the scripts in this way, the parent bash is not affected by erroneously programmed scripts or by any other problem that might affect the execution of the script. In addition, the PID of the child bash can be used as the "PID of the script" to kill it, stop it, etc. without affecting the parent bash (which the user might have used to execute other scripts). While this default behavior is convenient most of the times, there are some situations in which is not appropriate. For this purpose, there exists a shell built-in called `source`.

The `source` built-in allows executing a script without using any intermediate child bash. Thus, when we use `source` before the command-line of a script, we are indicating that the current bash must directly execute the commands of the script. Let's illustrate how `source` works. Firstly, you must create a *script*, which we will call `pids.sh`, as follows:

```
1 #!/bin/bash
2 echo PID of our parent process $PPID
3 echo PID of our process $$
4 echo Showing the process tree:
5 pstree $PPID
```

If you execute the script without `source`:

```
$ echo $$
2119
$ ./script.sh
PID of our parent process 2119
PID of our process 2225
Showing the process tree:
bash---bash---pstree
```

Now, if you execute the script with `source`:

```
$ source ./script.sh
PID of our parent process 2114
PID of our process 2119
Showing the process tree from PID=2114:
gnome-terminal---bash---pstree
```

As you observe, when script.sh is executed with `source`, the bash does not create a child bash but executes the commands itself. An alternative syntax for `source` is a single dot, so the two following command-lines are equivalent:

```
 $ source ./pids.sh
 $ . ./pids.sh
```

Now, we must explain which is the main utility of executing scripts with `source`. "Sourcing" is a way of including variables and functions in a script from the file of another script. This is a way of creating an environment but without having to "export" every variable or function. Using the example of the function `zip_contents()` of Section 11.8.1, we create a file called my_zip_lib.sh. This file will be our "library" and it will contain the function `zip_contents()` and the definition of a couple of variables:

```
1  #!/bin/bash
2  # file my_zip_lib.sh
3
4  OPTION="-l"
5  VAR="another variable..."
6
7  zip_contents() {
8    echo "Contents of $1: "
9    unzip $OPTION $1
10 }
```

Now, we can use '`source`' to make the function and the variables defined in this file available to our script.

```
1  #!/bin/bash
2  # script.sh
3
4  source my_zip_lib.sh
5
6  echo The variable OPTION from sourced from my_zip_lib.sh is: $OPTION
7  echo Introduce the name of the zip file:
8  read FILE
9  zip_contents $FILE
10 }
```

Note. If you don't use `source` to execute "my_zip_lib.sh" inside your script, you will not get the variables/functions available, since the child bash that is executing your script will create another child bash which is the one that will receive these variables/functions but that will be destroyed after "my_zip_lib.sh" is executed.

Position Variables

Position parameters are special variables that contain the arguments with which a script or a function is invoked. These parameters have already been introduced in Section 11.3. A useful command to manipulate position parameters is `shift`. This command moves to the left the list of parameters for a more comfortable processing of position parameters. This built-in and takes one argument, a number N. Then, the positional parameters are shifted to the left by this number N. In other words, the positional parameters from N+1 are renamed.

Special Variables

Special variables indicate the status of a process. They are treated and modified directly by the shell so they are read-only variables. In the above examples, we have already used most of them. For example, the variable $$ contains

Table 11.5: Special variables

Variable	Meaning
$$	Process PID.
$*	String with all the parameters received.
$@	Same as above but treats each parameter as a different word.
$#	Number of parameters.
$?	Exit status (or return code) of last command (0=normal, >0=error).
$!	PID of the last process executed in background.
$_	Value of last argument of the command previously executed.

the PID of the running process. The variable $$ is commonly used to assign names to files related to the process for temporary storage. As the PID is unique in the system, this is a way of easily identifying files related to a particular running process. Table 11.5 briefly summarizes these variables.

11.9 Extra

11.9.1 *Bash Builtins

A "builtin" command is a command that is built into the shell so that the shell does not fork a new process. The result is that builtins run faster and can alter the environment of the current shell. The shell will always attempt to execute the builtin command before trying to execute a utility with the same name. For more information on the builtins:

```
$ info bash
```

Table 11.6 shows bash builtins.

Table 11.6: *Bash builtins.*

Builtin	**Function**
:	returns exit status of 0
.	execute shell script from current process
bg	places suspended job in background
break	exit from loop
cd	change to another directory
continue	start with next iteration of loop
declare	display variables or declare variable
echo	display arguments
eval	scan and evaluate command line
exec	execute and open files
exit	exit current shell
export	place variable in the environment
fg	bring job to foreground
getopts	parse arguments to shell script
jobs	display list of background jobs
kill	send signal to terminate process
pwd	present working directory
read	read line from standard input
readonly	declare variable to be read only
set	set command-line variables
shift	promote each command-line argument
test	compare arguments
times	display total times for current shell and children
trap	trap a signal
umask	return value of file creation mask
unset	remove variable or function
wait [pid]	wait for background process to complete

11.9.2 *getops

When you want to parse commandline arguments in a professional way, `getopts` is the tool of choice. Unlike its older brother `getopt` (note the missing s!), it is a shell builtin command. The advantage is you do not need to hand your positional parameters through to an external program. `getopts` can easily set shell variables you can use for parsing, which is impossible for an external process. For further information consult the man of this builtin.

11.9.3 *Arrays

An array is a set of values identified under a unique variable name. Each value in an array can be accessed using the array name and an index. The built-in `declare -a` is used to declare arrays in bash. Bash supports single dimension arrays with a single numerical index but no size restrictions for the elements of the array. The values of the array can be assigned individually or in combination (several elements). When the special characters [] or [*] are used as index of the array, they denote all the values contained in the array. Next, we show the use of arrays with some examples:

```
#!/bin/bash
declare -a NUMBERS                      # Declare the array.
NUMBERS=( zero one two three )          # Compound assignment.
echo ${NUMBERS[2]}                      # Prints "two"
NUMBERS[4]="four"                       # Simple assignment.
echo ${NUMBERS[4]}                      # Prints "four"
NUMBERS=( [6]=six seven [9]=nine )      # Assign values for elements 6, 7 y 9.
echo ${NUMBERS[7]}                      # Prints "seven"
echo ${NUMBERS[*]}                      # Prints "zero one two three... nine"
```

11.9.4 *Associative Arrays

Bash provides one-dimensional indexed and associative array variables. While indexed arrays are referenced using integers (including arithmetic expressions); associative arrays are referenced using arbitrary strings. Associative arrays are created using declare -A name:

```
$ declare -A aa
$ aa[hello]=world
$ aa[ab]=cd
```

You can also assign multiple items at once:

```
$ declare -A aa
$ aa2=([hello]=world [ab]=cd)
```

The following expands to the list of keys (variables) of aa:

```
${!aa[@]}
```

The previous list can be used to access to each corresponding value in the associative array:

```
for VAR in ${!aa[@]}
  do
  echo $VAR
done
```

We can do more complex things with associative arrays like declare them at run time depending on the value of a variable. Since associative arrays are only one dimension, this can be used to create a kind of “matrix”. For example:

```
#!/bin/bash
rowname=expenses ; colname=flat
eval "declare -g -A matrix_$rowname"
eval "matrix_$rowname[$colname]=\"3600\""
eval "aux=matrix_$rowname[$colname]"
echo ${!aux}
```

The `-g` is to make the variable global (important to know that when used inside functions associative arrays are local by default). Note that to use a value of this array we need a temporary variable (`aux`). We can also build a list with variables that start with a prefix name:

```
for item in "${!row_*}"
do
echo $item
done
```

11.9.5 *More on parameter expansions

Simple usage

`$PARAMETER`
`${PARAMETER}`

Indirection

`${!PARAMETER}`

The referenced parameter is not PARAMETER itself, but the parameter named by the value of it. Example:

```
$ read -p "Which variable do you want to inspect?"
VAR
$ echo "The value of VAR is: ${!VAR}"
```

Case modification

`${PARAMETER^}` `${PARAMETER^^}` `${PARAMETER,}` `${PARAMETER,,}`

The `^` operator modifies the first character to uppercase, the `,` operator to lowercase. When using the double-form all characters are converted.

Variable name expansion

`${!PREFIX*}` `${!PREFIX@}`

This expands to a list of all set variable names beginning with the string PREFIX. The elements of the list are separated by the first character in the IFS-variable (<space> by default). Example:

```
$ echo ${!BASH*}
BASH BASH_ARGC BASH_ARGV BASH_COMMAND BASH_LINENO BASH_SOURCE ...
```

This will show all defined variable names (not values) beginning with BASH.

Substring removal

`${PARAMETER#PATTERN}` `${PARAMETER##PATTERN}` `${PARAMETER%PATTERN}` `${PARAMETER%%PATTERN}`

Expand only a part of a parameter's value, given a pattern to describe what to remove from the string. The operator "#" will try to remove the shortest text matching the pattern from the beginning, while "##" tries to do it with the longest text matching from the beginning. The operator "%" will try to remove the shortest text matching the pattern from the end, while "%%" tries to do it with the longest text matching from the end. Examples:

```
$ PATHNAME=/usr/bin/apt-get
$ echo ${PATHNAME##*/}
apt-get
$ echo ${PATHNAME#*/}
usr/bin/apt-get
```

Other examples:

```
$ FILE="tarball.tar.gz"
$ echo ${FILE%%.*}
tarball
$ echo ${FILE%.*}
tarball.tar
```

Search and replace

`${PARAMETER/PATTERN/STRING}`
`${PARAMETER//PATTERN/STRING}`

With one slash, the expansion only substitutes the first occurrence of the given pattern. With two slashes, the expansion substitutes all occurrences of the pattern. Example:

```
$ VAR="today is my day"
$ echo ${VAR/day/night}
tonight is my day
$ echo ${VAR//day/night}
tonight is my night
```

String length

`${#PARAMETER}`

The length of the parameter's value is expanded.

Substring expansion

`${PARAMETER:OFFSET}` cells `${PARAMETER:OFFSET:LENGTH}`

This one can expand only a part of a parameter's value, given a position to start and maybe a length.

Use a default value

`${PARAMETER:-string}`
`${PARAMETER-string}`

If the parameter PARAMETER is unset (never was defined) or null (empty), the first one expands to "string", otherwise it expands to the value of PARAMETER, as if it just was `${PARAMETER}`. If you omit the : (colon), like shown in the second form, the default value is only used when the parameter was unset, not when it was empty.

Assign a default value

`${PARAMETER:=string}`
`${PARAMETER=string}`

This one works like the using default values, but the default text you give is not only expanded, but also assigned to the parameter, if it was unset or null. Equivalent to using a default value, when you omit the : (colon), as shown in the second form, the default value will only be assigned when the parameter was unset.

Use an alternate value

`${PARAMETER:+string}`
`${PARAMETER+string}`

This form expands to nothing if the parameter is unset or empty. If it is set, it does not expand to the parameter's value, but to some text you can specify.

Display error if null or unset

`${PARAMETER:?string}`
`${PARAMETER?string}`

If the parameter PARAMETER is set/non-null, this form will simply expand it. Otherwise, the expansion of "string" will be used as appendix for an error message.

11.9.6 *SED

Introduction

`sed` is a stream editor. A stream editor is used to perform basic text transformations on an input stream (a file, or input from a pipeline). `sed` is very efficient because it only makes one pass over the input(s). `sed` maintains two data buffers (both are initially empty):

- active pattern space.
- auxiliary hold space.

In normal operation, `sed` reads in one line from the input stream. It removes any trailing newline and places the line in the pattern space. The pattern space is where text manipulations occur. A set of commands are applied to make the manipulations. Each command can have an address associated to it:

- An address is a kind of condition code.
- A command is only executed if the condition is verified.

When the end of the line is reached the contents of pattern space are printed out to the output stream. The trailing newline is added back if it was removed. Then, the next cycle starts for the next input line. The pattern space is deleted between two cycles, but by contrast, the hold space keeps its data between cycles. The hold space is initially empty, but there are commands for moving data between the pattern and hold spaces. Next, we show how `sed` works with some examples. These are simple examples using only the active pattern space.

Substitute Command

The substitute command changes all occurrences of the regular expression into a new value. A simple example is changing `day` in f1.txt to `night` in f2.txt:

```
$ sed s/day/night/ <f1.txt >f2.txt
```

If you have meta-characters in the command, quotes are necessary. And if you aren't sure, it's a good habit to use quotes:

```
$ sed 's/day/night/'<f1.txt >f2.txt
```

Another important concept is that, as mentioned, `sed` is line oriented. Suppose you have the input file:

```
one two three, one two three
four three two one
one hundred
```

If we use the command:

```
$ sed 's/one/ONE/' < f.txt
ONE two three, one two three
four three two ONE
ONE hundred
```

Note that this changed `one` to `ONE` once on each line. The first line had `one` twice, but only the first occurrence was changed. That is the default behavior. If you want something different, you will have to use some of the options that are available. On the other hand, there are four parts to this substitute command:

```
s          Substitute command
/../../    Delimiter
one        Search Pattern (Regular Expression)
ONE        Replacement string
```

The character after the `s` is the delimiter. It is conventionally a slash but it can be anything you want. If you want to use the delimiter you have to use the backslash to quote the delimiter.

Global replacement

If you tell `sed` to change a word, it will only change the first occurrence of the word on a line.

You may want to make the change on every word on the line instead of the first. If you want it to make changes for every match, add a `g` after the last delimiter:

```
$ sed 's/one/ONE/g' < f.txt
ONE two three, ONE two three
four three two ONE
ONE hundred
```

Using the Matched String

Sometimes you want to search for a pattern and add some characters to the matched string. For this purpose, you can use ”&“. Example:

```
$ sed 's/[a-z]*/(&)/' <old >new
```

The above command surrounds a string of any length with parentheses. You can have any number of & in the replacement string:

```
$ echo "123 abc" | sed 's/[0-9]*/& &/'
123 123 abc
```

Note that the first match of `[0-9]*` is a ZERO character at the beginning of the line. So if the input is `abc 123`, then the output is:

```
$ echo "abc 123" | sed 's/[0-9]*/& &/'
 abc 123
```

Note that the first character matched (ZERO) is replaced by a space in this example.

A better way to duplicate the number is to make sure it matches a number:

```
$ echo "abc 123" | sed 's/[0-9][0-9]*/& &/'
abc 123 123
```

Another possibility to write the previous command is to use extended regular expressions (e.g. the + sign). To enable them, you have to use the `-r` option of `sed`:

```
$ echo "abc 123" | sed -r 's/[0-9]+/& &/'
abc 123 123
```

Keep Part of the Pattern

The escaped parentheses (parentheses preceded by backslashes) store a substring of the characters matched by the regular expression. The `\1` is the first remembered pattern, the `\2` is the second remembered pattern and so on. `sed` has up to nine remembered patterns. For example, if you want to keep the first word of a line and delete the rest of the line:

```
$ sed 's/\([a-z]*\).*/\1/'
```

Regular expressions are greedy, and try to match as much as possible. `[a-z]*` matches zero or more lower case letters, and tries to match as many characters as possible. The `.*` matches zero or more characters after the first match. Since the first regex grabs all of the contiguous lower case letters, the second regex matches anything else:

```
$ echo abcd123 | sed 's/\([a-z]*\).*/\1/'
abcd
```

If you want to switch two words around, you can remember two patterns and change the order around:

```
$ sed 's/\([a-z]*\) \([a-z]*\)/\2 \1/'
```

Note that there is a space between the two remembered patterns. This is used to make sure two words are found. However, this will do nothing if a single word is found, or any lines with no letters. You may want to insist that words have at least one letter by using:

```
$ sed 's/\([a-z][a-z]*\) \([a-z][a-z]*\)/\2 \1/'
```

or by using extended regular expressions (note that parentheses no longer need to have a backslash):

```
$ sed -r 's/([a-z]+) ([a-z]+)/\2 \1/'
```

The `\1` does not have to be in the replacement string (in the right hand side) but it can be in the pattern you are searching for (in the left hand side). If you want to keep only the first occurrence of two consecutive duplicated words:

```
echo "hello hello how are you?" | sed 's/\([a-z]*\) \1/\1/'
hello how are you?
```

Print Only Some Lines

By default, `sed` prints every line. If it makes a substitution, the new text is printed instead of the old one. If you use `sed -n` it will not, by default, print any new lines. When the `-n` option is used, the `p` flag will cause the modified line to be printed. For example, if you want to print the lines with duplicated words:

```
$ sed -n '/\([a-z][a-z]*\) \1/p'
```

Another example is:

```
$ sed -n 's/pattern/&/p' <file
```

The above command is one way to duplicate the function of `grep` with `sed`.

11.9.7 *Debug Scripts

Debugging facilities are a standard feature of compilers and interpreters, and bash is no different in this regard. You can instruct bash to print debugging output as it interprets you scripts. When running in this mode, bash prints commands and their arguments before they are executed. The following simple script greets the user and prints the current date:

```
1 #!/bin/bash
2 echo "Hello $USER,"
3 echo "Today is $(date +'%Y-%m-%d')"
```

To trace the execution of the script, use bash -x to run it:

```
$ bash -x example_script.sh
+ echo 'Hello user1,'
Hello user1,
++ date +%Y-%m-%d
+ echo 'Today is 2011-10-24'
Today is 2011-10-24
```

In this mode, Bash prints each command (with its expanded arguments) before executing it. Debugging output is prefixed with a number of + signs to indicate nesting. This output helps you see exactly what the script is doing, and understand why it is not behaving as expected. In large scripts, it may be helpful to prefix this debugging output with the script name, line number and function name. You can do this by setting the following environment variable:

```
$ export PS4='+${BASH_SOURCE}:${LINENO}:${FUNCNAME[0]}: '
```

Let's trace our example script again to see the new debugging output:

```
$ bash -x example_script.sh
+example_script.sh:2:: echo 'Hello user1,'
Hello user1,
++example_script.sh:3:: date +%Y-%m-%d
+example_script.sh:3:: echo 'Today is 2011-10-24'
Today is 2011-10-24
```

Sometimes, you are only interested in tracing one part of your script. This can be done by calling `set -x` where you want to enable tracing, and calling `set +x` to disable it. Example:

```
#!/bin/bash
echo "Hello $USER,"
set -x
echo "Today is $(date %Y-%m-%d)"
set +x
```

You can run the script and you no longer need to run the script with bash -x. On the other hand, tracing script execution is sometimes too verbose, especially if you are only interested in a limited number of events, like calling a certain function or entering a certain loop. In this case, it is better to log the events you are interested in. Logging can be achieved with something as simple as a function that prints a string to stderr:

```
_log() {
  if [ "$_DEBUG" == "true" ]; then
    echo 1>&2 "$@"
  fi
}
```

Now you can embed logging messages into your script by calling this function:

```
_log "Copying files..."
cp src/* dst/
```

Log messages are printed only if the _DEBUG variable is set to true. This allows you to toggle the printing of log messages depending on your needs. You do not need to modify your script in order to change this variable; you can set it on the command line:

```
$ _DEBUG=true ./example_script.sh
```

Finally, if you are writing a complex script and you need a full-fledged debugger to debug it, then you can use `bashdb`, the Bash debugger.

11.9.8 *Expect

Introduction

`expect` is an automation scripting language that operates in much the same way humans do when interacting with a system. With `expect` you have to provide your script with commands and expected responses to those commands.

To install expect use your package manager. For example:

```
$ sudo apt-get install expect
```

Example

One of the simplest examples is to create an interactive SSH session. An script for `expect` could look like this:

```
#!/usr/bin/expect -f

spawn ssh alice.example.com
expect "password: "
send "YOURPASSW\r"
expect "$ "
send "ls -la\r"
expect "$ "
send "exit\r"
```

Let's explain it a little bit. The `-f` option tells `expect` that it is reading commands from a file (the script).

The spawn command launches an external command. In this case, `ssh` to a remote host called alice.example.com. When you do an `ssh`, you are prompted for a password. This password prompt is what you "expect" from the remote system; therefore, you enter that expected response. To send anything to the remote system, it must be included in double quotes and must include a hard return (\r). Once logged in the remote system, you can begin your interactive session on that remote host. Output will appear as STDOUT. After the command has executed, you are returned to a prompt, so you tell the `expect` script that bit of information:

```
expect "$ "
```

Finally, send the `exit` command to the remote system to log out.

11.10 Summary

Table 11.7 summarizes the commands used within this section and Table 11.8 summarizes the keywords and symbols used in shell scripts.

Table 11.7: Commands used in this section.

printf	format output.
dirname	given a pathname, extract its directory name.
basename	give a pathname, extract its base name.
bc	application for advanced math processing.

Table 11.8: Common keywords and symbols used in shell scripts.

$VAR	Value of variable VAR.
'	Quoting literals (without modifications).
"	Quoting except \ ` $.
`	*Command expansion*.
$1 ...	Positional parameters.
{X,Y,Z} {X..Y} {X..Y..Z}	Brace Expansion.
~ ~+ ~-	Tilde Expansion.
$PARAM ${PARAM...}	Parameter Expansion.
$(COMMAND) `COMMAND`	Command Substitution.
$((EXPRESSION)) $[EXPRESSION]	Arithmetic Expansion.
<(COMMAND) >(COMMAND)	Process Substitution.
*.txt page_1?.html	Filename Expansions.
*	For filename expansion, any string of characters, including a null (empty) string.
?	For filename expansion, any unique character.
[List]	For filename expansion, a unique character from the list.
[^List]	For filename expansion, a unique character not from the list.
`if [ expresssion ]; then; cmd; else; cmd; fi`	Basic conditional.
`for VARIABLE in list; do; cmd; done`	iteration statement.
`while [ expresssion ]; do; cmd; done`	another iteration statement.
`case string in ; pattern1); cmd ;; esac`	Conditional multiple.
`function function_name` or `function_name()`	define a function.
`let` or `declare -i`	Define a variable as numerical.
`local`	Define a variable local to a function.
`export` or `declare -x`	Define an exported variable.
`export -f`	Define an exported function.
`source`	Execute the script without a child bash.
`shift`	Move left the list of positional parameters.
$$	Process PID.
$*	String with all the parameters received.
$@	Same as above but treats each parameter as a different word.
$#	Number of parameters.
$?	Exit status (or return code) of last command (0=normal, >0=error).
$!	PID of the last process executed in background.
$_	Value of last argument of the command previously executed.

11.11 Practices

Exercise 11.1– Describe in detail line by line the following script:

```
#!/bin/bash
# AUTHOR: teacher
# DATE: 4/10/2011
# NAME: shellinfo.sh
# SYNOPSIS: shellinfo.sh [arg1 arg2 ... argN]
```

```
# DESCRIPTION: Provides information about the script.
# HISTORY: First version

echo "My PID is $$"
echo "The name of the script is $0"
echo "The number of parameters received is $#"

if [ $# -gt 0 ]; then
  I=1
  for PARAM in $@
  do
     echo "Parameter \$$I is $PARAM"
     ((I++))
  done
fi
```

Exercise 11.2– Describe in detail line by line the following script:

```
#!/bin/bash
# AUTHOR: teacher
# DATE: 4/10/2011
# NAME: clean.sh
# SYNOPSIS: clean.sh (without parameters)
# DESCRIPTION: Removes temporal files in your working directory:
# HISTORY: First version

echo "Really clean this directory?"
read YORN
case $YORN in
  y|Y|s|S) ACTION=0;;
  n|N) ACTION=1;;
  *) ACTION=2;;
esac

if [ $ACTION -eq 0 ]; then
   rm -f  \#* *~ .*~ *.bak .*.bak *.backup *.tmp .*.tmp core a.out
   echo "Cleaned"
   exit 0
elif [ $ACTION -eq 1 ]; then
   echo "Not Cleaned"
   exit 0
elif [ $ACTION -eq 2 ]; then
   echo "$YORN is no an allowed answer. Bye bye"
   exit 1
else
    echo "Uaggg!! Symptomatic Error"
    exit 2
fi
```

Exercise 11.3– Develop a script that calculates the square root of two cathetus. Use a function with local variables and arithmetic expansions.

Exercise 11.4– Describe in detail line by line the following script files:

```
#!/bin/bash
# AUTHOR: teacher
# DATE: 4/10/2011
# NAME: fill_terminal_procedure.sh
# SYNOPSIS: fill_terminal arg
# DESCRIPTION: Procedure to fill the terminal with a printable character
# FUNCTION NAME: fill_terminal:
# OUTPUT: none
# RETURN CODES: 0-success 1-bad-number-of-args 2-not-a-printable-character.
# HISTORY: First version

fill_terminal() {

[ $# -ne 1 ] && return 1

   local HEXCHAR DECCHAR i j
   HEXCHAR=$1
   DECCHAR=`printf "%d" 0x$HEXCHAR`
   if [ $DECCHAR -lt 33  -o $DECCHAR -gt 127 ]; then
     return 2
   fi
   [ -z "$COLUMNS" ] && COLUMNS=80
   [ -z "$LINES" ] && LINES=24
   ((LINES-=2))
   for((i=0; i< COLUMNS; i++))
   do
     for ((j=0; j< LINES; j++))
     do
       printf "\x$HEXCHAR"
     done
   done
   return 0
}
```

```
#!/bin/bash
# AUTHOR: teacher
# DATE: 4/10/2011
# NAME: procedure.sh
# SYNOPSIS: procedure.sh arg
# DESCRIPTION: Use the fill_terminal procedure
# HISTORY: First version

source fill_terminal_procedure.sh
fill_terminal $@
case $? in
  0)
  exit 0 ;;
  1)
  echo "I need one argument (an hex value)" >&2 ; exit 1 ;;
  2)
  echo "Not printable character. Try one between 0x21 and 0x7F" >&2 ; exit 1 ;;
```

```
  *)
  echo "Internal error" >&2 ; exit 1
esac
```

Exercise 11.5– The following script illustrates how to use functions recursively. Describe it in detail line by line.

```
#!/bin/bash
# AUTHOR: teacher
# DATE: 4/10/2011
# NAME: recfind.sh
# SYNOPSIS: recfind.sh file_to_be_found
# DESCRIPTION: Search recursively a file from the working directory
# HISTORY: First version

# Function: search_in_dir
# Arguments: search directory
function search_in_dir() {
  local fileitem
  [ $DEBUG -eq 1 ] && echo "Entrant a $1"
  cd $1
  for fileitem in *
  do
    if [ -d $fileitem ]; then
       search_in_dir $fileitem
    elif [ "$fileitem" = "$FILE_IN_SEARCH" ]; then
       echo `pwd`/$fileitem
    fi
  done
  [ $DEBUG -eq 1 ] && echo "Sortint de $1"
  cd ..
}

DEBUG=0

if [ $# -ne 1 ]; then
  echo "Usage: $0 file_to_search"
  exit 1
fi

FILE_IN_SEARCH=$1

search_in_dir `pwd`
```

Exercise 11.6– Using a function recursively, develop a script to calculate the factorial of a number.

Exercise 11.7– In this exercise, we will practice with Regular Expressions (regex).

1. Create a file called re.txt and type a command-line that continuously ”follows“ this file to display text lines added to this file in the following way:

   Display only the text lines containing the word ”kernel“.

   From these lines, display only the first 10 characters.

   Try your command-line by writing from another pseudo-terminal some text lines to the file re.txt.

Hint: use `tail`, `grep` and `cut`.

Note. You must use the `grep` command with the option `--line-buffered`. This option prevents `grep` from using its internal buffer for text lines. If you do not use this option you will not see anything displayed on the terminal.

2. Type a command-line that continuously ”follows“ the re.txt file to display the new text lines in this file in the following way:

   Display only the text lines ending with a word containing three bowels.

   Try your command-line by sending from another pseudo-terminal some text lines to re.txt.

Chapter 12

System Administration

12.1 Users Management

12.1.1 User Accounts

An account provides the user with configuration settings and preferences and typically also with some space in disk (normally under the `/home` directory). More generically, we can find different types of users (or accounts):

- Superuser, administrator or `root` user. This user has a special account which is used for system administration. The `root` user has granted all the rights over all the files and processes.
- Regular users. A user account provides access to the system with a limited access to critical resources such as files and directories.
- Special users. The accounts of special users are not used by human beings but they are used by internal system services. An example is teh user www-data, which is used by Web servers to access to documents and resources. In your system, some services migth be executed by `root`. However, in general, the use of special users is preferred due to security reasons.

12.1.2 Configuration Files

The configuration information of the users is saved in the following files and directories:

`/etc/passwd` — text file containing user account information

`/etc/shadow` — text file containing user password information

`/etc/group` — text file containing groups

`/etc/gshadow` — text file that may contain group passwords

`/etc/skel` — directory that contains files and directories that are automatically copied over to a new user's home directory when such user is created by the `useradd` command.

The previous text files can be edited with any editor but it is recommneded in case you need to do so, to use an editor like `vipw` and `vigr` so that the file cannot be edited while others are changing it.

Example text lines of the file `/etc/passwd` could be the following:

```
telematics:x:1000:1000:Mike Smith,,,:/home/telematics:/bin/bash
root:x:0:0:root:/root:/bin/bash
```

Note. If it appears :: this means that the corresponding field is empty.

The fields have the following meaning:

1. `telematics`: this field contains the name of the user, which must be unique within the system.
2. `x`: this field contains the encoded password. The "x" means that the password is not in this file but it is in the file `/etc/shadow`.
3. `1000`: this field is the number assigned to the user, which must be unique within the system.
4. `1000`: this field is the number of the default group. The members of the different groups of the system are defined in the file `/etc/group`.
5. `Mike Smith`: this field is optional and it is normally used to store the full name of the user. Some user creation commands such `adduser` add comas because The comment field in the passwd file is seen as a GECOS field with several subfields. These subfields are the user's full name (or application name, if the account is for a program), the building and room number or contact person, the office telephone number and any other contact information (pager number, fax, etc.). However, these subfields are not generally much used.
6. `/home/telematics`: this field is the path to the home directory.
7. `/bin/bash`: this field is the default shell assigned to the user.

On the other hand, the `/etc/shadow` file essentially contains the encrypted password liked with the user name and some extra information about the account. A sample text line of this file is the following:

```
telematics:a1gNcs82ICst8CjVJS7ZFCVnu0N2pBcn/:12208:0:99999:7:::
```

Where we can observe the following fields:

1. User name : It is your login name
2. Password: It your encrypted password. The password should be minimum 6-8 characters long including special characters/digits
3. Last password change (lastchanged): Days since Jan 1, 1970 that password was last changed
4. Minimum: The minimum number of days required between password changes i.e. the number of days left before the user is allowed to change his/her password
5. Maximum: The maximum number of days the password is valid (after that user is forced to change his/her password)
6. Warn : The number of days before password is to expire that user is warned that his/her password must be changed
7. Inactive : The number of days after password expires that account is disabled
8. Expire : days since Jan 1, 1970 that account is disabled i.e. an absolute date specifying when the login may no longer be used

The file `/etc/shadow` must be readable only by `root`. This is to avoid other people getting a copy of all the hashed passwords and running crack program (like John the Ripper) to try to recover passwords.

Another important configuration file related with user management is `/etc/group`, which contains information of system groups. A sample text line of this file is the following:

```
users:x:1000:telematics,otheruser
```

Where the fields follow this format:

```
group-name:password-group:ID-group:users-list
```

The users-list is optional since this information is already stored in the `/etc/passwd` file.

On the other hand, groups can also have an associated password (although this is not very common because shared passwords are insecure). Typically, the encrypted password of the group is stored in another file called `/etc/gshadow`.

Finally, the directory `/etc/skel` contains the "skeleton" that is copied to each user's home when the user is created.

12.1.3 Creating a User Account

Being **root** on a Linux system, you can manually create a new user account by following the next steps:

1. Find the next available UID and GID numbers, or use the ones provided, checking they are unique.

2. Add an entry to the `/etc/passwd` and `/etc/shadow` files. This includes a hash of the password into `/etc/shadow`.

3. Create the home directory (under /home).

4. Create a mail spool file `/var/spool/mail/username`.

5. Copy the files and directories from `/etc/skel` to the home directory.

6. Change the ownership of the home directory and all its contents to the user, and the group ownership to the primary group of the user.

7. Change the ownership of the mail spool file to the user, and make the group owner equal to `mail`.

Fortunately, we have a command called `useradd` (there is also another similar command called `adduser`) that does all the previous steps.

12.1.4 Management Commands

Below, there is a list of commands related to management of users. These commands modify the files explained in the previous section.

- `useradd`: add a new user.
- `userdel`: delete a user.
- `usermod`: modify a user (e.g. add it to a group). However, be careful because it removes the user from any groups not specified. To add a user to a group you can use instead `gpasswd -a user group`.
- `groupadd`, `groupdel`, `groupmod`: management of groups.
- `passwd`: change your password. If you are root you can also change the password of other users.
- `su`: switch user (change of user). Go to section 12.1.6 for further information.
- `who`: shows who is logged in the system and their associated terminals.
- `last`: shows a list of last logged users.
- `id`: prints the real and effective user and group IDs.
- `groups`: prints the groups which the user belongs to.
- `chage`: change and list user password expiry information (e.g. `chage -l`).
- `chown`: can be used to change the owner and the group of a file.
- `chgrp`: can be used to change the group of a file.

Examples of the `chown` command:

```
# chown telematics notes.txt
```

The above command sets a new owner (telematics) for the file `notes.txt`. Only the superuser can change the owner of a file and users can change the group of a file. Obviously, a user can only change the group of a file to a one which she belongs. The "-R" option is commonly used to make the change recursive. Example:

```
# chown -R student.mygroup directory
```

The command above sets the owner to "student" and the group to "mygroup" of the directory and all its contents.

Finally, you can suspend ("lock") an account by inserting an exclamation mark '`!`' in front of the password field in /etc/shadow using `vipw` or you can use `sudo passwd -l username` to do the same thing. You can unlock the account by removing the '`!`' either manually with `vipw` or with `sudo passwd -u username`.

12.1.5 Special Accounts

Special users have accounts that generally have a user ID that is lower than some particular value. This value is recommended by each Linux distribution to avoid unintentional conflicts.

These accounts are not intended for human beings so they typically cannot access to a login shell. This is accomplished using `/bin/false` or `/sbin/nologin` and also disabling the password.

Example configuration line of FTP User in /etc/passwd:

```
ftp:x:14:50:FTP User:/var/ftp:/sbin/nologin
```

12.1.6 sudoers File

As explained in Section 1.5, the `sudo` command uses the sudoers file (/etc/sudoers) to determine if a user can execute a certain command with certain privileges. In the sudoers file, system administrators can define which users or groups will be able to execute certain commands (or even any command) as root but these users will not have to know the password of the root. The `sudo` command **prompts to introduce the password of the user that is executing `sudo`**. In this way, the command `sudo` makes it easier to implement the principle of "least privilege". It also logs all commands and arguments so there is a record of who used it for what, and when (in /var/log/auth.log). Next, we provide an example of a sudoers file (/etc/sudoers):

```
# This file MUST be edited with the 'visudo' command as root.
# See the man page for details on how to write a sudoers file.
Defaults        env_reset
# Host alias specification
# User alias specification
User_Alias NET_USERS = %netgroup
# Cmnd alias specification
Cmnd_Alias NET_CMD =/usr/local/sbin/simtun, /usr/sbin/tcpdump, /bin/ping,
/usr/sbin/arp, /sbin/ifconfig, /sbin/route, /sbin/iptables, /bin/ip
# User privilege specification
root    ALL=(ALL) ALL
# Uncomment to allow members of group sudo to not need a password
# (Note that later entries override this, so you might need to move
# it further down)
# %sudo ALL=NOPASSWD: ALL
NET_USERS       ALL=(ALL)       NOPASSWD: NET_CMD
# Members of the admin group may gain root privileges
%admin ALL=(ALL) ALL
```

In the previous example, all the users belonging the group "netgroup" can access some network-related commands. In addition, the users belonging to the "admin" group have access to any command. Due to the complexity of the sudoers file and the possible problems of a wrong configuration (loosing access, etc.), it is not recommended to edit the sudoers file with a regular editor but with a special editor like `visudo`.

Next, we provide a couple of examples of sudo (take a look also at Section 1.5). For example, to edit `index.html` as the user `www-data`:

```
$ sudo -u www-data vi /var/run/www/htdocs/index.html
```

To execute several commands in a subshell with sudo and make redirection work:

```
$ sudo sh -c "cd /home ; du -s * | sort -rn > output.txt"
```

12.2 Special File Permissions

Three special types of permissions: **setuid**, **setgid** and **sticky bit** are available for executable files and public directories. These permissions are activated adding an octal number to the access mask: setuid (1000), setgid (2000) and sticky bit (4000). You can also add the setuid with u+s or remove it with u-s:

```
$ chmod u+s script
$ chmod u-s script
```

You can also add the setgid with g+s or remove it with g-s:

```
$ chmod g+s myscript
$ chmod g-s myscript
```

Finally, you can add/remove the sticky bit of a directory with +t and -t:

```
# chmod +t /home/user1/data
# chmod -t /home/user1/data
```

12.2.1 setuid

When the set user identification (setuid) permission is set on an executable file, a process that runs this file is granted access based on the owner of the file (typically root), rather than the user who is running the executable file. This special permission allows a user to access files and directories that are normally only available to the owner. To test this permission we can use the following C program:

```
#include <stdio.h>
#include <unistd.h>

/* test_program */
main(){
  printf("UID: %d, EUID: %d\n",getuid(),geteuid());
}
```

Every process has a real user identifier (UID) and an Effective UID (EUID). The real UID is the UID of the user account that starts the process. The effective UID is the UID of the user account whose privileges attach to the process. In the general case, the real and effective UIDs are the same, setuid allows you to alter that behavior. To this respect, the previous program shows the UID and EUID while the process is being executed. So, let's compile and execute this `test_program`:

```
user1$ gcc -o test_program test_program.c
user1$ chmod 755 test_program
user1$ ./test_program
UID: 1000, EUID: 1000
```

If you test the program with the setuid set and with another user (user2 with uid 1001):

```
user2$ ./test_program
UID: 1001, EUID: 1000
```

You must be extremely careful when you set the setuid permission, because special permissions constitute a security risk. A critical example that can allow a regular user to gain superuser privileges is when executing a program owned by root that has set the user ID (UID). Also, regular users can set special permissions for files they own, which constitutes another security concern.

In your system, you should monitor for any unauthorized use of the setuid and setgid permissions to gain superuser privileges or to set not desired privileges.

12.2.2 setgid

The set-group identification (setgid) permission is similar to setuid, except that the process's effective group ID (GID) is changed to the group owner of the file, and a user is granted access based on permissions granted to that group.

When setgid permission is applied to a directory, files created by process in a directory with gid belong to the group of the directory, not the group to which the process belongs. Any user who has write and execute permissions

in the directory can create a file there. However, the file belongs to the group of the directory, not to the user's group ownership.

12.2.3 Sticky Bit

The sticky bit is a permission bit that protects the files within a directory. If the directory has the sticky bit set, a file can be deleted only by the owner of the file, the owner of the directory, or by root. This special permission prevents a user from deleting other users' files from public directories such as /tmp.

12.3 System Clock

12.3.1 Cron

Cron is a daemon/service that executes shell commands periodically on a given schedule. Cron is driven by a crontab, a configuration file that holds details of what commands are to be run along with a timetable of when to run them. You can see a list of active crontab entries by entering the following terminal command:

```
$ crontab -l
```

You can create a crontab file by entering the following terminal command:

```
$ crontab -e
```

The previous command opens a text editor with a new blank crontab file, or it will open your existing crontab if you already have one. The cron file for each user is stored at `/var/spool/cron/crontabs/user`. However, the cron file is not thought to be directly accessed by users (in fact, users do not have rights to directly modify this file).

Each crontab line has six fields (separated by white spaces):

- **m** (0-59): representing the minute of the hour.
- **h** (0-23): representing the hour of the day.
- **dom** (1-31): representing the day of the month.
- **mon** (1-12): representing the month of the year.
- **dow** (0 - 6 ; sunday=0): representing the day of the week.
- **command**: which is the command to be run, exactly as it would appear on the command line.

A double-ampersand "`&&`" can be used to run multiple commands consecutively.
For the numbers you can use:

- A list of numbers, which is a set of integers separated by commas, for example 15,30,45, which would represent just those three numbers.
- A range of numbers, which is a set of numbers separated by a hyphen, for example 10-20, which would represent all the numbers from 10 through 20, inclusive.
- You can mix the previous: "0-4,8-12".
- The asterisks ("*") in our entry tell cron that for that unit of time, the job should be run "every".
- The slash "/" is to specify a step value. This can be used in conjunction with ranges. For example, you can write "0-23/2" in Hour field to specify that some action should be performed every two hours (it will have the same effect as "0,2,4,6,8,10,12,14,16,18,20,22"). Steps are also permitted after an asterisk, so if you want to say "every two hours", you can use "*/2".

Examples:

```
* * * * * <command> #Runs every minute
30 * * * * <command> #Runs at 30 minutes past the hour
45 6 * * * <command> #Runs at 6:45 am every day
45 18 * * * <command> #Runs at 6:45 pm every day
00 1 * * 0 <command> #Runs at 1:00 am every Sunday
00 1 * * 7 <command> #Runs at 1:00 am every Sunday
00 1 * * Sun <command> #Runs at 1:00 am every Sunday
30 8 1 * * <command> #Runs at 8:30 am on the first day of every month
00 0-23/2 02 07 * <command> #Runs every other hour on the 2nd of July
```

As well as the above there are also special strings that can be used:

```
@reboot <command> #Runs at boot
@yearly <command> #Runs once a year [0 0 1 1 *]
@annually <command> #Runs once a year [0 0 1 1 *]
@monthly <command> #Runs once a month [0 0 1 * *]
@weekly <command> #Runs once a week [0 0 * * 0]
@daily <command> #Runs once a day [0 0 * * *]
@midnight <command> #Runs once a day [0 0 * * *]
@hourly <command> #Runs once an hour [0 * * * *]
```

By default a cron job will send an email to the user account executing the cronjob. If this is not needed put the following command at the end of the cron job line:

```
>/dev/null 2>&1
```

If you already have a crontab file, you can use it with the following command:

```
$ crontab -u <username> <crontab file>
```

To remove your crontab file simply enter the following terminal command:

```
$ crontab -r
```

12.3.2 At

If you need to do a "one time command scheduling" you can use the `at` command. In this sense it is complementary to cron which usually is used to schedule periodic jobs. Example:

```
$ at 10:27 ls -l
warning: commands will be executed using /bin/sh
job 2 at Tue May 5 10:27:00 2015
```

The output will be mailed to you (at your local user account) after it runs. You can use the `atq` to list the `at` commands in que and what time it is scheduled for.

12.4 Start Up Applications

Many times, to administrate your Linux system, it is necessary to run a script on a system when it boots up. There are several possibilities to do so.

12.4.1 rc.local

You can execute your application after the system boots up by putting it in /etc/rc.local. The applications in this directory are executed while Linux is booting up and before any users have logged in. When rc.local executes, the path environment variable (or any other environment variable) may not have been set up so it is important to use the program or script's full path. It's also important to make sure the program does not prompt for user input, as users

won't have logged in when rc.local runs. If the program waits for user input, it won't complete, and the boot process will grind to a halt before users can login. Make sure /etc/rc.local is executable and in your script use a shebang with sh like `#!/bin/sh -e`

Example:

```
#!/bin/sh -e
# rc.local
#
# This script is executed at the end of each multiuser runlevel.
# Make sure that the script will "exit 0" on success or any other
# value on error.
# In order to enable or disable this script just change the execution
# bits.
# By default this script does nothing.

sleep 10
/bin/echo "testing" > /tmp/rc_local_test.log
exit 0
```

To test that everything works fine:

```
$ sudo /etc/init.d/rc.local start
```

12.4.2 crontab

You can use `@reboot` to tell cron to run a task when your system boots.

12.4.3 init

In /etc/init.d/ you can create your own script for the service or services that you want to run at boot time. A base script to construct a init.d script can be found in /etc/init.d/skeleton. Once you create your script, say `myservice`, you can install it on the system with the following command:

```
$ sudo update-rc.d /etc/init.d/myservice defaults
```

When Linux boots up or shuts down, it looks in these folders to see if any scripts or programs need to be run. To remove a service:

```
$ sudo update-rc.d -f /etc/init.d/myservice remove
```

You can also disable a service:

```
$ sudo update-rc.d /etc/init.d/myservice disable
```

12.4.4 Display Server and Window Managers

To start graphical applications you can use configuration files of your display server. If your display server is X, the typical files are:

- The file ~/.xinitrc, which is meant for when you start Xorg with the startx command.
- If you are running a display manager instead, you will need an ~/.xsession script instead.

Finally, you can also configure startup scripts and commands after your window manager starts. You should take a look at the documentation of your particular window manager.

12.5 System Logging

12.5.1 Introduction

One great ability of Unix-like systems is to “log” events that happen. Some possible things you may see logged are kernel messages, system events, user runs any program, user runs a particular program, user calls a specific system function, user su’s to root, server events etc.

As with any piece of software in Linux, you have options as to which system logger program you would like to use. Some popular ones include: syslog, metalog, sysklogd and rsyslog. All of these are good choices, but we are going to work with rsyslog, which is the default logger program in Ubuntu.

12.5.2 Log Locations

The location of the various log files varies from system to system. On most UNIX and Linux systems the majority of the logs are located in /var/log. Some of these log files are:

Table 12.1: Common Linux log files name and usage

/var/log/syslog	General message and system related stuff
/var/log/auth.log	Authenication logs
/var/log/kern.log	Kernel logs
/var/log/boot.log	System boot log
/var/log/wtmp	Login records file
/var/log/daemons.log	Logs for network daemons
/var/log/apache2/	Apache 2.0 access and error logs directory

In short, /var/log is the location where you should find all Linux logs file. However some applications such as `apache2` (one of the most famous WEB servers) have a directory within /var/log/ for their own log files.

12.5.3 Logrotate

`logrotate` is a great utility to manage the log files in a system. It provides you with a lot of options, namely, automatic rotation, compression and mailing of log files. Each log file can be rotated hourly, daily, weekly or monthly. The application renames old files and redirects logging to a new file. For example, `messages` → `messages.1` → `messages.2` → `messages.3` → `messages.4` → delete.

`logrotate` is typically run under `cron`. The main configuration file is `/etc/logrotate.conf` but many configuration belongs to the software packages, which put a file into directory `/etc/logrotate.d/`.

```
# see "man logrotate" for details
# rotate log files weekly
weekly

# keep 4 weeks worth of backlogs
rotate 4

# create new (empty) log files after rotating old ones
create

# uncomment this if you want your log files compressed
#compress

# packages drop log rotation information into this directory
include /etc/logrotate.d

# system-specific logs may be configured here
```

12.5.4 Kernel Log (dmesg)

All UNIX and Linux systems have a log that is actually part of the kernel. In practice the log is actually a section of memory in the kernel used to record information about the kernel that may be impossible to write to disk because the information is generated before the filesystems are loaded.

For example, during the boot process, the filesystems are not accessible for writing (most kernels boot with the filesystem in read mode until the system is considered safe enough to switch to read/write mode). The data in this

log contains information about the devices connected to the system and any faults and problems recorded by the system during the boot and operational process. In some systems the information is periodically dumped into a file (/var/log/dmesg) but the most fresh information is only available using the `dmesg` command (for most UNIX/Linux variants).

12.5.5 System Logs

The rsyslog service is a daemon that runs the background and accepts log entries and writes them to one or more individual files. All messages reported to syslog are tagged with the date, time, and hostname. In addition, it is possible to have a single server that accepts log messages from a number of hosts, writing out the information to a single file. The service is highly configurable (in our case in /etc/rsyslog.conf and /etc/rsyslog.d). In these files you must specify rules. Every rule consists of two fields, a selector field and an action field. These two fields are separated by one or more spaces or tabs. The selector field specifies a pattern of facilities and priorities belonging to the specified action.

12.5.6 Selectors

The selector field consists of two parts: a facility and a priority.

- The facility is one of the following keywords:
  auth, authpriv, cron, daemon, kern, lpr, mail, mark, news, syslog, user, uucp and local0 through local7.
- The priority defines the severity of the message and it is one of the following keywords, in ascending order:
  debug, info, notice, warning, err, crit, alert, emerg.

Additionally, the following keywords and symbols have a special meaning: “none”, “*”, “=” and “!”. We show their use by examples below.

12.5.7 Actions

The action field of a rule describes the abstract term “logfile”. A “logfile” need not to be a real file but it can be also a named pipe, a virtual console or a remote machine. To forward messages to another host, prepend the hostname with the at sign “@”.

12.5.8 Examples

```
*.info;mail.none;news.none;authpriv.none;cron.none    /var/log/syslog
```

The *.info means “Log info from all selectors”. However, after that, it says mail.none;news.none and so forth. What that means when all put together is “Log everything from these EXCEPT these things that are following it with ’.none’ behind them”.

```
*.crit;kern.none          /var/adm/critical
```

The configuration above will store all messages with the priority crit in the file /var/adm/critical, except for any kernel message.

```
# Kernel messages are first, stored in the kernel
# file, critical messages and higher ones also go
# to another host and to the console
#
kern.*                          /var/adm/kernel
kern.crit                       @mylogserver
kern.=crit                      /dev/tty5
kern.info;kern.!err             /var/adm/kernel-info
```

- The first rule directs any message that has the kernel facility to the file /var/adm/kernel.

- The second statement directs all kernel messages of the priority crit and higher to the remote host *mylogserver*. This is useful, because if the host crashes and the disks get irreparable errors you might not be able to read the stored messages.

- The third rule directs kernel messages of the priority crit to the virtual console number five (`tty5`).

- The fourth saves all kernel messages that come with priorities from info up to warning in the file /var/adm/kernel-info. Everything from err and higher is excluded.

To activate the rsyslog service in a remote server, edit the /etc/rsyslog.conf file adding:

```
$ModLoad imudp
$UDPServerRun 514
```

And restart rsyslog:

```
# service rsyslog restart
```

12.5.9 Other Logging Systems

We must remark that many programs deal with their own logging. Apache Web server is one of those. In your httpd.conf file you must specify where you are logging things.

12.5.10 Logging and Bash

To view log files you can user the `tail -f` command. Example:

```
# tail -f /var/log/syslog
```

On the other hand, the `logger` command makes entries in the system log and it provides an interface with the log system for shells. Examples:

To log a message indicating a system reboot, enter:

```
telematics$ logger System rebooted
telematics$ tail -1 /var/log/syslog
Sep  7 11:28:02 XPS telematics: System rebooted
```

To log a message contained in the /tmp/msg1 file, enter:

```
$ logger -f /tmp/msg1
```

To log the news facility critical level messages, enter:

```
$ logger -p news.crit Problems in our system
```

12.6 Creating Filesystems

In the next sections we will show you how to mount and unmount filesystems, how to create partitions and how to create the initial filesystem configuration for your system.

12.6.1 Mounting a Filesystem

The command to mount a storage device under a mount point is `mount`. For example, if our first SATA disk has a second partition, the Kernel will map it in /dev/sda2. Then, we can "mount" some part of your filesystem in this second partition. For example, let us consider that we want to store the /home directory in this second partition of sda, for this you can type the following command:

```
# mount /dev/sda2 /home
```

In this example, /dev/sda2 is the device and /home is the mount point. The mounting point can be defined as the directory under which the contents of a storage device can be accessed. Linux can also mount WINDOWS filesystems such as fat32:

```
# mount -t vfat /dev/hdd1 /mnt/windows
```

This will mount a `vfat` file system (Windows 95, 98, XP) from the first partition of the `hdd` device in the mount point `/mnt/windows`.

12.6.2 Unmounting a Filesystem

After using a certain device, we can "unmount" it. For example, if the file system of a pen-drive mapped in /dev/sdc1 is mounted under the directory or mount point `/media/pen`, then any operation on `/media/pen` will act in fact over the FS of the pen-drive. When we finish working with the pen-drive, it is important to "unmount" the device before extracting it. This is because unmouting gives the opportunity to the OS of finishing all I/O pending operations. This can be achieved with the command `umount` using the mount point or the device name[1]:

```
# umount /media/pen
```

or

```
# umount /dev/sdc1
```

Furthermore, when `/media/pen` is unmounted, then all I/O operations over `/media/pen` are not performed over the pen anymore. Instead, these operations are performed over device that is currently mounting the directory, usually the system's hard disk that is mounting the root ("/").

Note. It is not possible to unmount an "busy" (in use) file system. A file system is busy if there is a process using a file or a directory of the mounted FS.

12.6.3 Create Partitions and Filesystems

The main command to view and create partitions is `fdisk`. For example, to display list of partitions:

```
# fdisk -l /dev/sdb
```

Note. # means that the command must be executed as *root*.

If you want to create partitions on the device /dev/sdb you must type:

```
# fdisk /dev/sdb
```

And then, follow the menu instructions. To make the changes available you should type:

```
# partprobe /dev/sdb
```

Once we have created the desired partitions, we can create a filesystem with the command `mkfs`.

For example:

[1] Today you can also unmount a mount point using the menus of the GUI.

```
# mkfs.ext4 /dev/sdb2
```

The previous command creates an ext4 filesystem in the second partition of the device /dev/sdb.

On the other hand, if our partition is corrupted in some way we can check and repair it with the command `fsck`. However, to execute this command the partition to be checked/repaired must not be mounted.

Another interesting tool is `gparted`, which is a graphical application that allows to create, remove, view, mount, unmount and format partitions. Finally, it is worth to mention that when you install a Linux system majority of the distributions include some tool to manage the partitions and filesystems that will be created in the system.

12.6.4 Backup MBR

The Master Boot Record (MBR) of your hard drive, for example /dev/sda, is stored in the first 512 bytes and includes the bootstrap code and partition table. You can Backup your MBR using:

```
# dd if=/dev/sda of=/tmp/myMBR.bak bs=512 count=1
```

If the MBR were damaged, it would be necessary to boot to a rescue disk and use the following command, which performs essentially the reverse operation of the one above:

```
# dd if=/tmp/myMBR.bak of=/dev/sda
```

Notice that it is not necessary to specify the block size and block count as in the first command because the dd command will simply copy the backup file to the first sector of the hard drive and stop when it reaches the end of the source file.

12.6.5 Loop Disk

For testing purposes, we are going to show how to use a regular file as a disk. The example presented in this section is interesting to understand how the file systems in some virtual machines work. Next, we show this process step by step:

1. Create the file that will contain the filesystem filled with zeros (in our case a file of 100 MB called "/var/disk").

   ```
   # dd if=/dev/zero of=/var/disk count=1 bs=100M
   ```

2. We will need a special loopback device so we look for one not currently used in the system.

   ```
   # losetup /dev/loop0
   ```

   Replace /dev/loop0 with /dev/loop1, /dev/loop2, etc, until a free loopback device is found. Attach the loopback device (e.g. /dev/loop0) with the file:

   ```
   # losetup /dev/loop0 /var/disk
   ```

3. Next, we create a filesystem (e.g. ext4) using the loopback device:

   ```
   # mkfs.ext4 /dev/loop0
   ```

4. Create a mount directory and mount the filesystem:

   ```
   # mkdir /mnt/myfs
   # mount /dev/loop0 /mnt/myfs
   ```

5. Finally, if you want to unmount the loopback file system and release the loopback device, type:

   ```
   # umount /mnt/myfs
   # losetup -d /dev/loop0
   ```

Now, all the Linux filesystem-related commands can be executed on this "new" file system. For example, you can type `df -h` to confirm its disk usage.

Finally, you can create partitions on the /dev/loopX devices. For example, if we create partitions in /dev/loop0, the partitions are called loop0p1, loop0p2, etc.

12.6.6 File fstab

To create the initial filesystem, we need some way of specifying which storage devices and which mount points are used during booting. On the other hand, only the root user can mount filesystems as this is a risky operation, so we also need a way of defining mount points for unprivileged users. This is necessary for instance, for using storage devices like pen-drives or DVDs.

The fstab file provides a solution to the previous issues. The fstab file lists all available disks and disk partitions, and indicates how they are to be initialized or otherwise integrated into the overall system's file system. fstab is still used for basic system configuration, notably of a system's main hard drive and startup file system, but for other uses (like pen-drives) has been superseded in recent years by "automatic mounting".

The fstab file is most commonly used by the mount command, which reads the fstab file to determine which options should be used when mounting the specified device. It is the duty of the system administrator to properly create and maintain this file.

An example of an entry in the /etc/fstab file is the following:

```
/dev/sda1   /   ext4   defaults   1 1
```

The 1st and 2nd columns are the device and default mount point. The 3rd column is the filesystem type. The 4th column are mount options and finally, the 5th and 6th columns are options for the `dump` and `fsck` applications. The 5th column is used by `dump` to decide if a filesystem should be backed up. If it's zero, dump will ignore that filesystem. This column is zero many times. The 6th column is a `fsck` option. fsck looks at the number in the 6th column to determine in which order the filesystems should be checked. If it's zero, `fsck` won't check the filesystem.

12.6.7 Persistent block device naming

A device name like /dev/sdb1 is based on where your physical drive is plugged in and the order the drives were made available to the computer, so if your computer changes the same command could mount a different partition. It's possible for this to happen just from a software upgrade. There are four different schemes for persistent naming: by-label, by-uuid, by-id and by-path. In particular, we discuss by-label and by-uuid (by-id and by-path are much less used in practice).

Labels

Almost every filesystem type can have a label. All your partitions that have a label are listed in the /dev/disk/by-label directory. This directory is created and destroyed dynamically, depending on whether you have partitions with labels attached. The labels of your filesystems can be changed. Following are some methods for changing labels on common filesystems:

```
swap:       swaplabel -L <label> /dev/XXX using util-linux
ext2/3/4 :  e2label /dev/XXX <label> using e2fsprogs
btrfs:      btrfs filesystem label /dev/XXX <label> using btrfs-progs
```

For example, to view the current label in an EXT filesystem:

```
# e2label /dev/sda1
```

To set a new label, enter:

```
# e2label /dev/sdb2 data1
```

Then, you can use the label in /etc/fstab:

```
#device          mount point      FS       Options Backup  fsck
LABEL=data1      /mnt/data1       ext3     _netdev 0       0
```

UUID

UUID is a mechanism to give each filesystem a unique identifier. These identifiers are generated by filesystem utilities (e.g. mkfs.*) when the partition gets formatted and are designed so that collisions are unlikely. All GNU/Linux filesystems support UUID. FAT and NTFS filesystems do not support UUID, but are still listed in /dev/disk/by-uuid with a shorter UID (unique identifier). A UUID will remain the same if you put an internal disk into an external USB caddy, or change the name of the partition.

You can type `ls -al /dev/disk/by-uuid/` to see UUID entries. The advantage of using the UUID method is that it is much less likely that name collisions occur than with labels. Further, it is generated automatically on creation of the filesystem. Then, you can use the UUID in /etc/fstab:

```
#device          mount point      FS       Options Backup  fsck
UUID=9467f4de-4231-401f-bcaa-fee718d49e85 /   ext4     errors=remount-ro 0 0
```

In general, UUIDs are automatically created and it is extremely rare to end with two exact UUIDs in a system. The exception is when you clone a partition (for example using the `dd` command). In this case, you create an exact copy of the partition all the way down to the UUID and you end with 2 partitions with the same UUID. From the example of my /etc/fstab above, the UUID is no longer unique and it will mount the first partition it finds with that UUID. In this case, you need to change the UUID. To do so, first you have to find the device path using the following command:

```
# sudo blkid
/dev/sda1: UUID="aabe7e48-2d11-421f-8609-7ea9d75e7f9b" TYPE="swap"
/dev/sda2: UUID="9467f4de-4231-401f-bcaa-fee718d49e85" TYPE="ext4"
/dev/sdb1: UUID="9467f4de-4231-401f-bcaa-fee718d49e85" TYPE="ext4"
```

Here you can see that /dev/sda2 and /dev/sdb1 have the same UUID. Let's generate a new UUID:

```
$ uuidgen
f0acce91-a416-474c-8a8c-43f3ed3768f9
```

Finally apply the new UUID to the partition:

```
$ sudo tune2fs /dev/sdb1 -U f0acce91-a416-474c-8a8c-43f3ed3768f9
```

12.6.8 User Mounts

There are several ways of letting a user to mount a device. A way is to configure /etc/fstab. However, this is a static solution and difficult to implement mounts with removable devices. Another is to allow the mount command through sudo. However, this is very insecure. A few potential situations are:

- The user could mount a filesystem with a suid root copy of bash and then, running that instantly gives root (likely without any logging, beyond the fact that mount was run).
- A user could mount his own filesystem on top of /etc, containing his/her own copy of /etc/shadow or /etc/sudoers, then obtain root with either su or sudo. Or possibly bind-mount (mount –bind) over one of those two files. Or a new file into /etc/sudoers.d.
- Similar attacks could be pulled off over /etc/pam.d and many other places.

The various desktop environments have actually already built solutions to this, to allow users to mount removable media. They work by mounting in a subdirectory of /media only and by turning off set-user/group-id support via kernel options. There are several applications that can do this. In particular, next, we describe `pmount` and `udisks`.

Let's start with `pmount`. Most distributions do not install `pmount` out of the box but it is a quick download from most package managers. Once installed, there is some configuration that may need to be done before a use can utilize the `pmount` command. The /etc/pmount.allow file must be created listing all the possible devices that will be mountable using `pmount`. In addition, `pmount` will mount **removable devices** (removable devices in general are not listed in /etc/pmount.allow). To determine if a device is flagged as removable issue this command:

```
$ cat /sys/block/[device]/removable
```

Where [device] is the device on your system:

- If the value is "1" then it is removable and, the device does not need a white-list entry.
- If it is "0" then it is not considered a removable device and must be white-listed to be used with `pmount`.

Devices are specified in the while-list one per line. By default all devices are mounted under the /media directory by their given partition name. So issuing:

```
$ pmount /dev/sdb1
```

Will mount partition 1 of device sdb on /media/sdb1. You can also pass a label to pmount and the partition or device will be mounted under that label:

```
$ pmount /dev/sdb clipzip
```

To find out what partitions are available for mounting when you plug a drive into your system you can issue the `dmesg` command which will show you the device name and the partitions on the device. If execute `pmount` without parameters, it will show all the devices mounted by the `pmount` command (if there are any), just like issuing the mount command. Once a device is mounted use the `pumount` command to umount it:

```
$ pumount /dev/sdb
```

Another useful command for user mounts is `udisks`. This is the modern replacement for gnome-mount (but in fact, it is not gnome specific). When a disc (e.g. USB stick) is mounted under your file browser (e.g. nautilus) it uses `udisks` behind the scenes. You can do the same thing on the command line with the `udisks` command. Example:

```
$ udisks --mount /dev/sdb1
```

The parameter after `--mount` is the device name of the partition you want to mount. The command will mount /dev/sdb1 in /media/<uuid> where <uuid> is the identifier of the particular partition.

You can also use the UUID to perform the mount:

```
$ udisks --mount /dev/disk/by-uuid/1313-F422
```

Again, this will mount your partition in /media/<uuid> which is not consistent with how nautilus mounts partitions. The partitions mounted by nautilus can be found in /media/<user>/<uuid> with <user> being the current logged-in user. To keep the folder structure consistent an alternative command can be used that takes care of the correct mount point automatically:

```
$ udisksctl mount --block-device /dev/disk/by-uuid/<uuid>
```

To unmount you can use the option `--unmount`.

12.7 Extra

12.7.1 *Quotes

If you manage a system that's accessed by multiple users, you might have a user who hogs the disk space. Using disk quotas you can limit the amount of space available to each user. It's fairly easy to set up quotas, and once you are done you will be able to control the number of inodes and blocks owned by any user or group.

Control over the disk blocks means that you can specify exactly how many bytes of disk space are available to a user or group. Since inodes store information about files, by limiting the number of inodes, you can limit the number of files users can create.

When working with quotas, you can set a soft limit or a hard limit, or both, on the number of blocks or inodes users can use. The hard limit defines the absolute maximum disk space available to a user. When this value is reached, the user can't use any more space. The soft limit is more lenient, in that the user is only warned when he exceeds the limit. He can still create new files and use more space, as long as you set a grace period, which is a length of time for which the user is allowed to exceed the soft limit.

```
$ quotatool -bq 15000M -l "15010 Mb" -u user /home
```

12.7.2 *Accounts Across Multiple Systems

In an environment composed of multiple systems, users would like to have a single login name and password. A tool is required to efficiently perform user management in this type of environments. Examples of such tools are NIS (Network Information System), NIS+, and LDAP (Lightweight Directory Application Protocol). These tools use large databases to store information about users and groups in a server. Then, when the user logs into a host of the environment, the server is contacted in order to check login and configuration information.

This allows having a single copy of the data (or some synchronized copies), and users can access to hosts and resources from different places. These tools also include additional concepts such as hierarchies or domains/zones for accessing and using hosts and resources.

12.7.3 *Access Control Lists

Access Control Lists (ACLs) provide a much more flexible way of specifying permissions on a file or other object than the standard Unix user/group/owner system. Access Control Lists (ACLs) allow you to provide different levels of access to files and folders for different users. One of the dangers that ACLs attempt to avoid is allowing users to create files with 777 permissions, which become system wide security issues. For instance, ACLs will allow you to set a file where one user can read, other users cannot read and yet other users are able to read and write to the same file. This is not possible with the standard permission system of Unix. ACLs however, tend to be more complicated to maintain than the standard permission system. In Linux, to use ACLs we just need to mount the file system with proper options. ACL are available for most file systems and current kernels are compiled by default with the ACL extension. To mount a partition with ACL we have to include the acl option. Example in the fstab:

```
/dev/sda1      /        ext4      defaults,acl    1 1
```

The process of changing ACLs is fairly simple but sometimes understanding the implications is much more complex. There are a few commands that will help you make the changes for ACLs on individual files and directories.

```
$ getfacl file
```

The `getfacl` command will list all of the current ACLs on the file or directory. For example if *user1* creates a file and gives ACL rights to another user (*user2*) this is what the output would look like:

```
$ getfacl myfile
# file: myfile
# owner: user1
# group: user1
user::rw-
user:user2:rwx
group::rw-
mask::rwx
other::r--
```

The `getfacl` shows typical ownership as well as additional users who have been added with ACLs like *user2* in the example. It also provides the rights for a user. In the example, *user2* has rwx to the file myfile. The `setfacl`

command is used to create or modify the ACL rights. For example, if you wanted to change the ACL for *user3* on a file you would use this command:

```
$ setfacl -m u:user3:rwx myfile
```

The -m is to modify the ACL and the "u" is for the user which is specifically named, *user3*, followed by the rights and the file or directory. Change the "u" to a "g" and you will be changing group ACLs.

```
$ setfacl -m g:someusers:rw myfile
```

If you want to configure a directory so that all files that are created will inherit the ACLs of the directory you would use the "d" option before the user or group.

```
$ setfacl -m d:u:user1:rw directory
```

To remove rights use the "x" option.

```
$ setfacl -x u:user1 file
```

12.8 Command Summary

The table 12.2 summarizes the commands used within this section.

Table 12.2: *Commands.*

`mount`	mounts a filesystem.
`umount`	unmounts a filesystem.
`fdisk`	create and show partitions.
`mkfs`	create filesystems.
`dd`	disk duplicate or create a low level copy of a file system.
`gparted`	graphically manage partitions and filesystems.
`useradd`	adds user to the system (onluy root).
`userdel`	deletes user from the system (only root).
`usermod`	modifies user (only root).
`groupadd`	adds a group to the system (only root).
`groupdel`	deletes a group (only root).
`groupmod`	modifies group info (only root).
`passwd`	changes user password.
`su`	changes user.
`sudo`	changes user for executing a certain command.
`who`	shows logged in users.
`last`	shows system last access.
`id`	shows user ids, groups...
`groups`	shows system groups.
`chown`	changes file owner.
`chgrp`	changes file group.

12.9 Practices

Exercise 12.1– This exercise deals with system management. To do the following practices you will need to be "root". To not affect your host system, we will use an UML virtual machine.

1. Start an UML Kernel with a filesystem mapped in the `ubda` device (of the UML guest) and with 128 Megabytes of RAM. Then, login as *root* in the UML guest and use the command `uname -a` to view the Kernel versions of the host and the UML guest. Are they the same? explain why they can be different.

2. Login as *root* in the UML guest and type a command to see which is your current uid (user ID), gid (group ID) and the groups for which *root* is a member. Type another command to see who is currently logged in the system.

3. As root, you have to create three users in the system called *user1*, *user2* and *user3*. In the creation command, you do not have to provide a password for any of the users but you should create them with "home" directories and with a Bash as default *shell*.

4. Type the proper command to switch from *root* to *user1*. Then, type a command to view your new uid, gid and the groups for which *user1* is a member. Also find these uid and gid in the appropriate configuration files.

5. Type a command to see if currently *user1* appears as logged in the system. Explain which steps do yo need to follow to login as *user1*. Once you manage to login as *user1*, type a command to view who is currently connected, type another command to view which are the last login accesses to the system and finally, type another command to view your current uid, gid and groups.

6. Login as *user1* and then switch the user to *root*. As *root* create a group called "someusers". Find the gid assigned this group. Modify the configuration of *user1*, *user2* and *user3* to make them "**additionally**" belong to the group "someusers". Type "exit" to switch back to *user1*, and then, type a command to view your current uid, gid and groups. Do you observe something strange?

7. Logout and login again as *root*. Switch to *user1*, and then, type a command to view your current uid, gid and groups. Explain what you see. Switch back to *root* and create another group called "otherusers". Then, modify *user2* and *user3* so that they **additionally** belong to this group.

8. Now, we are going to simulate that our UML guest has a second disk or storage device. First, halt the UML guest. In the host, create an ext3 filesystem of 30M called second-disk.fs. We are going to map second-disk.fs in the 'ubdb" device of the UML guest. To do so, add the option "ubdb=second-disk.fs" to the command line that you use to start the UML guest.

9. Login as root in the UML guest. As you can see, the new filesystem (ubdb) is not mounted so mount it under /mnt/extra. Type a command to view the mounted filesystems and another command to see the disk usage of these filesystems. Inside /mnt/extra, create a directory called "etc" and another directory called "home".

10. Modify the /etc/fstab configuration file of the UML guest so that the mount point /mnt/extra for the device ubdb is automatically mounted when booting the UML guest. Reboot the UML guest and check that /dev/ubdb has been automatically mounted.

11. Halt the UML guest. In the host, mount the filesystem second-disk.fs with the loop option (use any mount point that you want). Then, copy the file /etc/passwd of your host into the "etc" directory of the filesystem of second-disk.fs you have mounted. Now, unmount second-disk.fs in the host and boot again the UML guest. Login as root in the UML guest and check that the file passwd exists in /mnt/extra/etc.

12. Now, we are going to simulate that we attach an USB pen-drive into our UML guest. To do so, you have to create a new filesystem of 10 Megabyte called pen-drive.fs **inside** the UML guest. Then, mount pen-drive.fs with the loop option in /mnt/usb. Now, we are going to use our "pen-drive": create a subdirectory in /mnt/usb called "etc" and then unmount /mnt/usb. Next, mount again pen-drive.fs with the loop option but this time in /mnt/extra. Why is now the directory /mnt/extra/etc empty? Copy the file /etc/passwd of the UML guest in /mnt/extra/etc and then umount. Which are now the contents of the /mnt/extra/etc file? why?

13. Type the following commands and explain what they do:

```
guest# mkdir /mnt/extra/root2
guest# mount /dev/ubda /mnt/extra/root2
guest# ls /mnt/extra/root2
guest# chroot /mnt/extra/root2
guest# mount
guest# ls /mnt/extra
guest# exit
guest# umount /mnt/extra/root2
```

14. Login as root in the UML guest and create a configuration in the system for the directory /mnt/extra in which the three users: *user1*, *user2* and *user3* can create and delete files in this directory. Check the configuration.

15. Now, create a configuration for the directory /mnt/extra so that *user1* and *user2* can create and delete files in /mnt/extra but *user3* can only list the contents of this directory. Check the configuration.

16. Create three subdirectories called user1, user2 and user3 in /mnt/extra. These directories have to be assigned to their corresponding users (*user1*, *user2* and *user3*). Then, create a configuration in the UML guest for these subdirectories so that each user can only create and delete files in her assigned subdirectory but can only view the contents of the subdirectory of another user. Finally, users in the system different from *user1*, *user2* and *user3* must not have access to these subdirectories.

17. As root, create and execute a Bash script called "links.sh" that creates a symbolic link in the home of each user to the directory /mnt/extra.

18. Add a symbolic link to /mnt/extra in the directory /etc/skel. Then, create a new user called *user4* without password but with a home and Bash as *shell* and explain what you see in the home of this user.

19. Type a command to write ”probing logging system“ in the general system log (the /var/log/syslog file in Debian-like distributions). Then, type a command to view the last 10 lines of /var/log/syslog.

Part V

Appendices

Appendix A

Ubuntu in a Pen-drive

A.1 Install

In this appendix we show how to install Ubuntu in an USB pen-drive. Once the system has been installed, we configure properly the system in order to extend as much as possible the life of the USB device.

Important note. This installation procedure is only valid for x86 architectures, which means that it is not valid for MAC computers.

Let' start with the process. First, we must insert the Unbutu CD and configure the BIOS of our computer to boot from the CD unit. Next, we have to choose the language for installation, then select "Install Ubuntu" and choose the language for the OS (Figure A.1)

Then, we select the time zone and the keyboard layout (Figure A.2).

Now, we must select the disk and the partition in which we want to install the OS.
Now, we must select to specify disk partitions manually (advanced).

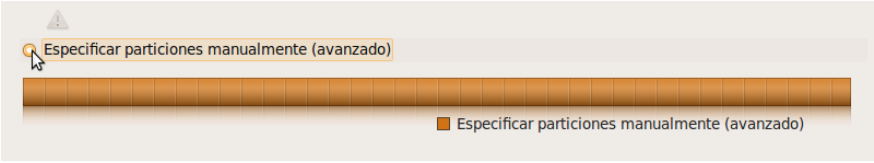


Figure A.3: Advanced specification of disk partitions

We must properly identify the disk device

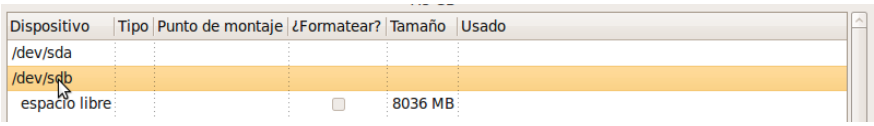


Figure A.4: Advanced specification of disk partitions 2

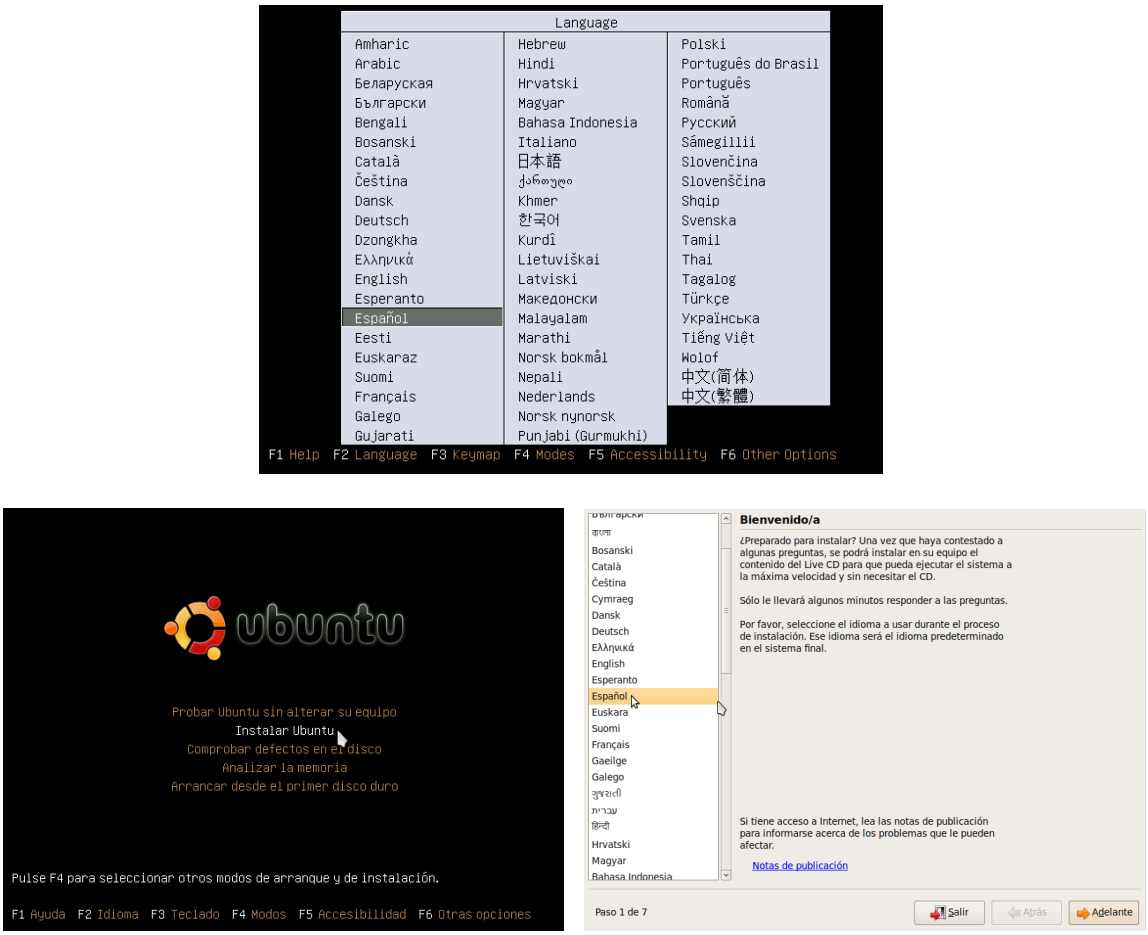


Figure A.1: First selection windows

In this sample installation /sda is our SATA hard disk so the disk partition selected is /sdb, which is the device of our USB pen-drive. **Note. The size of disks can help us to select correctly the device of the pen-drive.**

The next step requires filling in the fields for our account (see Figure A.5). **Note. This account will be by default in the "sudoers" file with administration capabilities.**

Figure A.2: Time zone and Keyboard layout

This step is tricky, because if we make a mistake when selecting the disk/partition we can cause data loss in the system.

Figure A.5: Account Information

The next step allows us to import documents and settings for programs (like Firefox) or other OS (like Windows) already installed in the disk units. In this case, we will press "Next" as we do not want to import anything.

Figure A.6: Import documents and settings

We are at the last step before starting the installation. This window reports all the parameters we have set. However, it also has an option called “Advanced”. We must type this button.

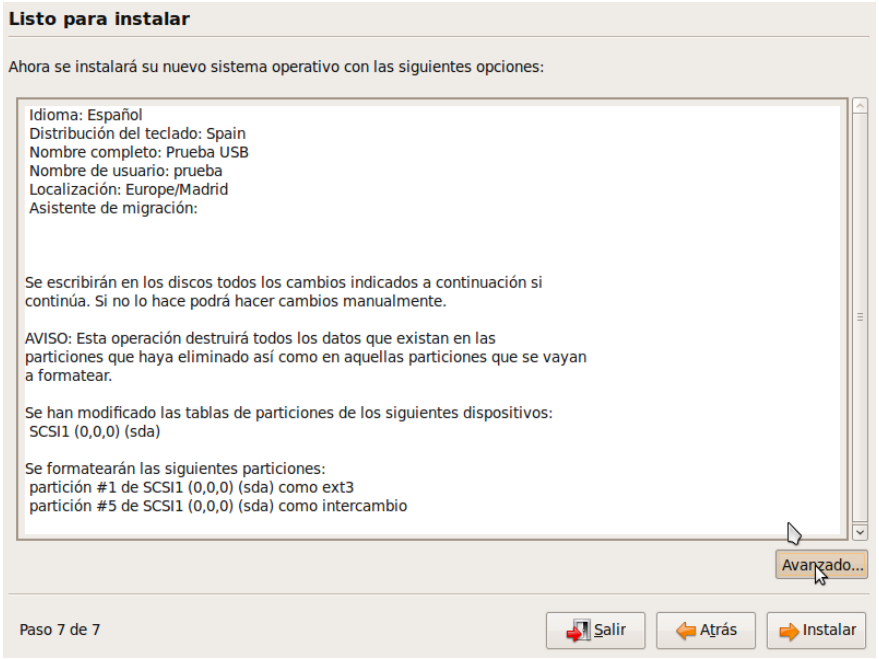


Figure A.7: Final installation step

In this window we have the possibility of selecting where we want to install the boot loader (Grub2 in the current Ubuntu version). In this case, we simply need to specify /dev/sdb, which is the device in which we are doing the installation.

Figure A.8: Selecting the partition of the Boot loader

GRUB (GRand Unified Bootloader) is a multi boot manager used in many "Linux distros" to start several operating systems within the same computer. **It is very important to correctly make this final step because with this window we modify the boot sector of the corresponding hard disk (USB pen-drive) called MBR (Master Boot Record) in order to properly point to the boot loader code. If we miss this step we will install the boot loader over the defalt disk, probably /dev/sda and the computer will not be able to boot neither from its main disk (/dev/sda) nor from the USB pen-drive (/dev/sdb).**

Once the installation is complete, we might have to modify our BIOS settings to select the USB device as primary boot device.

A.2 Tunning the system

Flash drives and solid state drives are shock resistant, consume less power than traditional disks, produce less heat, and have very fast seek times. However, these type of disks have a more limited total number of write operations than traditional disks. Fortunately, there are some tweaks you can make to increase performance and extend the life of these type of disks.

- The simplest tweak is to mount volumes using the noatime option. By default Linux will write the last accessed time attribute to files. This can reduce the life of your disk by causing a lot of writes. The noatime mount option turns this off. Ubuntu uses the relatime option by default. For your disk partitions, replace relatime with noatime and nodiratime in /etc/fstab. You can also add the option commit=120 to set to 120 seconds the pedding data writes:

```
/dev/sda1 / ext4 noatime,nodiratime,commit=120,errors=remount-ro 0 0
```

- Another tweak is using a ramdisk instead of a physical disk to store temporary files. This will speed things up and will protect your disk at the cost of a few megabytes of RAM. We will make this modification in our system so, edit your /etc/fstab file and add the following lines:

```
tmpfs      /tmp          tmpfs      defaults          0    0
tmpfs      /var/tmp      tmpfs      defaults          0    0
```

  This lines tell the Kernel to mount these temporary directories as tmpfs, a temporary file system. We will avoid unnecessary disk activity and in addition when we halt the system we will automatically get rid of all temporal files.

- Reduce “swap” operations that write to disk. Change the value to a lower number

```
sudo sysctl -w vm.swappiness=0
```

  Paste `vm.swappiness=5` in /etc/sysctl.conf. Reboot and check the setting again to confirm the changes were made in /proc/sys/vm/swappiness.

- The final tweak is related with our web browser. If you use Firefox, this browser puts its cache in your home partition. By moving this cache in RAM you can speed up Firefox and reduce disk writes. Complete the previous tweak to mount /tmp in RAM, and you can put the cache there as well. Open `about:config` in Firefox. Right click in an open area and create a new string value called `browser.cache.disk.parent_directory` and set the value to /tmp (see Figure A.9). When we reboot the system all the previous changes must be active.

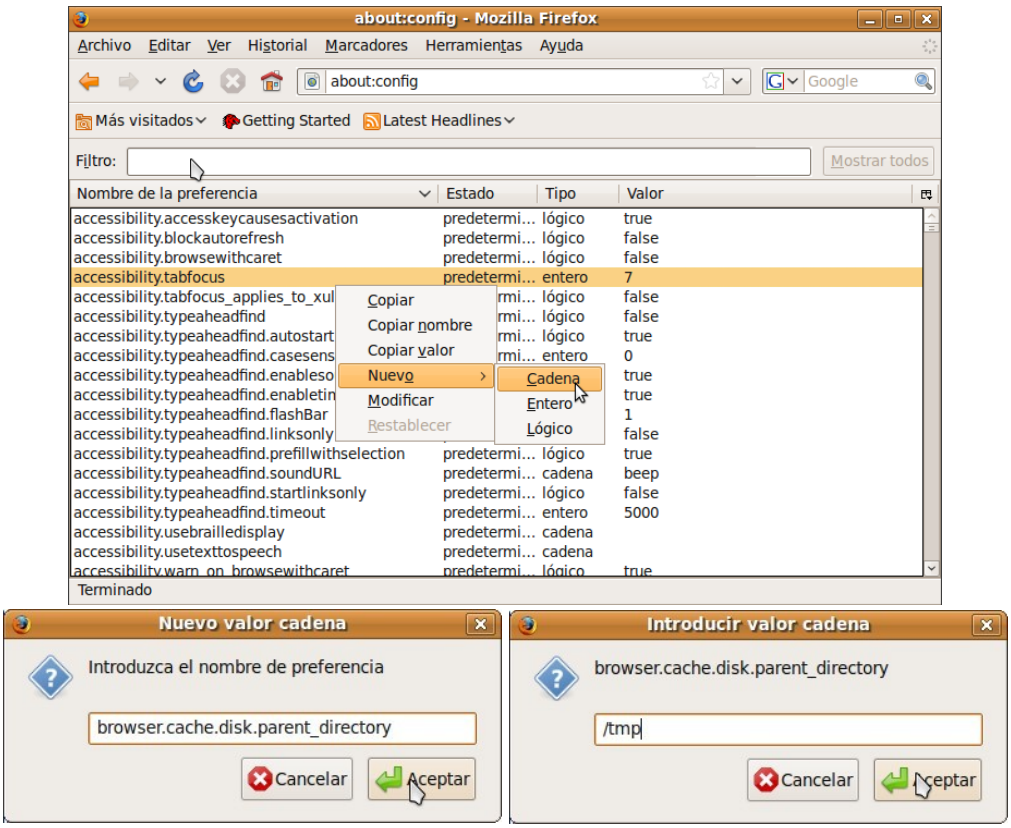


Figure A.9: Firefox cache in /tmp