Jose L. Muñoz
Juanjo Alins
Jorge Mata
Oscar Esparza
Carlos H. Gañán

UPC Telematics Department

Linux Networks

Contents

Part I

Linux Essentials

Chapter 1

Introduction to Unix/Linux

Chapter 2

Introduction to Unix/Linux

2.1 Introduction to OS

The main goal of this chapter is understanding the our environment: the Linux Operating System. In short, an Operating System (OS) is a set of programs whose purpose is to provide a way of managing resources of a computer system and also to provide an interface for interaction with human beings. Computer resources to be managed by an OS include CPU, RAM, and I/O devices. These I/O devices include secondary storage devices (HDD, CD, DVD, USB, etc.) and communication devices like wired networking (Ethernet) or wireless networking (Wifi), etc. From the above discussion, we can say that two key issues for an OS are:

- **Resources management**. 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 the RAM memory required by each process. An introduction to which is the organization of the Linux OS which allows this OS to achieve a proper way of managing and accessing system resources is provided in Section 2.2.
- **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 2.3.

2.2 Resources Management

Today, 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 a process[1] or task at a time (also called mono-process or mono-task) and to manage only a user (mono-user). On the other hand, UNIX was designed from the very beginning as a multi-user system. This has numerous advantages, even for a system where only one or two people will be using it. Security, which is necessary for protection of sensitive information, is built into UNIX at selectable levels. More importantly, the system was designed also from the very beginning to multi-task. Whether one user is running several programs or several users are running one program, UNIX systems were capable of managing the situation. As shown in Figure 2.1, today both type 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 WINDOWS.

On the other hand, modern OS can be divided in at least three parts: hardware, kernel and user applications (also called user space). See Figure 2.2.

[1] A process is basically a running program.

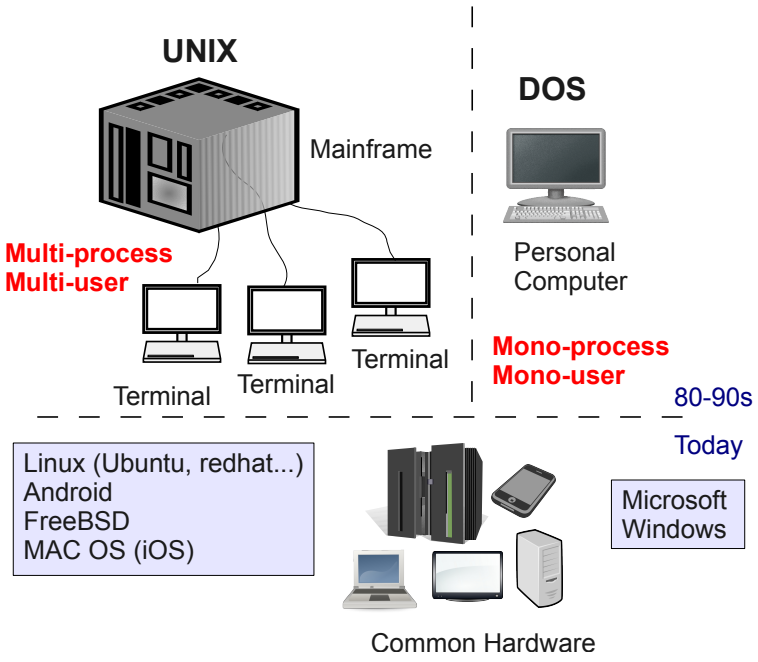


Figure 2.1: Origins of Linux.

The kernel is the main component of most computer operating systems. It is a bridge between 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. Typically, the resources consist of:

- **Central Processing Unit (CPU**). This is the most central part of a computer system, responsible for running or executing programs on it. The kernel takes responsibility for deciding at any time which of the many running programs should be allocated to the processor or processors (each of which can usually run only one program at a time).

- **Memory.** Memory is used to store both program instructions and data. Typically, both need to be present in memory in order for a program to execute. Often multiple programs will want access to memory, frequently demanding more memory than the computer has available. The kernel is responsible for deciding which memory each process can use, and determining what to do when not enough is available.

- **Input/Output (I/O) Devices.** I/O devices present in the computer, such as keyboard, mouse, disk drives, printers, displays, etc. The kernel allocates requests from applications to perform I/O to an appropriate device (or subsection of a device, in the case of files on a disk or windows on a display) and provides convenient methods for using the device (typically abstracted to the point where the application does not need to know implementation details of the device).

The kernel typically makes these facilities available to application processes through system calls (see Figure 2.3).

A system call defines how a program requests a service from an operating system's kernel. This may include hardware related services (for e.g. accessing the Hard Disk), creating and executing new processes, and communicating with integral kernel services (like scheduling). System calls provide the interface between a process and the operating system. Most operations interacting with the system require permissions not available to a user level process, e.g. I/O performed with a device present on the system, or any form of communication with other processes requires the use of system calls.

The Linux Kernel a monolithic hybrid kernel. Drivers and kernel extensions typically execute in a privileged zone known as "Ring 0" (see again Figure 2.2), with unrestricted access to hardware. Unlike traditional monolithic kernels,

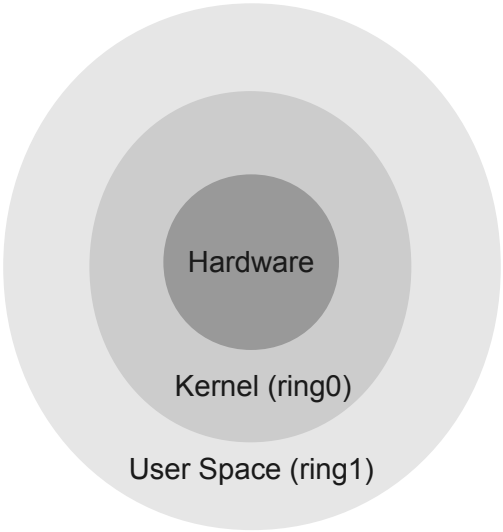


Figure 2.2: OS Rings.

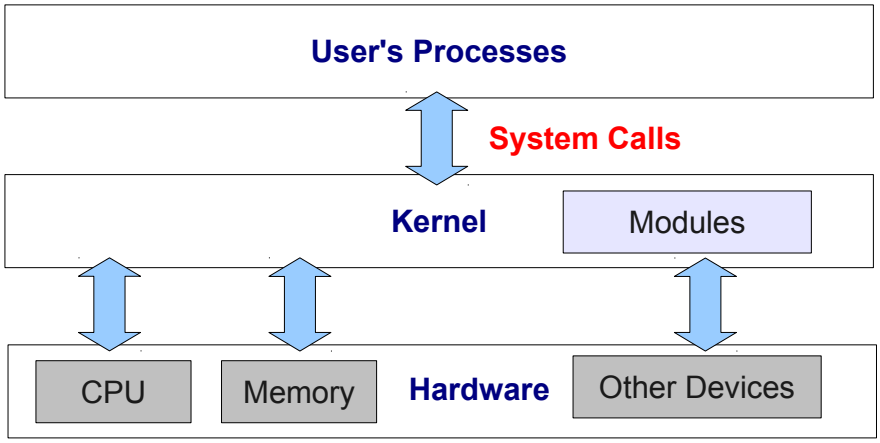


Figure 2.3: Modular Design.

drivers and kernel extensions can be loaded and unloaded easily as modules. Modules are pieces of code that can be loaded and unloaded into the kernel upon demand. They extend the functionality of the kernel without the need to reboot the system. For example, one type of module is the device driver, which allows the kernel to access hardware connected to the system. Without modules, we would have to build monolithic kernels and add new functionality directly into the kernel image. Besides having larger kernels, this has the disadvantage of requiring us to rebuild and reboot the kernel every time we want new functionality. Modules or drivers interact with devices like hard disks, printers, network cards etc. and provide system calls. Modules can be statically compiled with the kernel or they can be dynamically loaded. The commands related with modules are `lsmod`, `insmod` and `modprobe`.

Finally, we will discuss the Linux Booting Sequence. To start a Linux system (see Figure 2.4), we need to follow a startup sequence in which the flow control goes from the BIOS to the boot manager [2] and finally to the Linux-core (kernel).

As mentioned, Unix-like systems are multiprocess and to manage the processes the kernel starts a scheduler and executes the initialization program *init*. *init* sets the user environment and allows user interaction and the log in, then

[2]The boot loader stage is not absolutely necessary. Certain BIOS can load and pass control to Linux without the use of the loader. Each boot process will be different depending on the architecture processor and the BIOS.

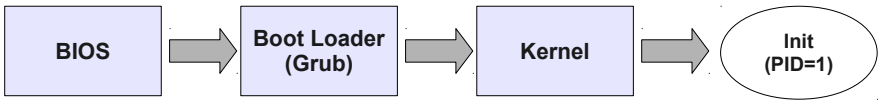


Figure 2.4: The Linux Boot Process.

the kernel keeps itself idle until it is called. In summary, after you boot Linux:

- The kernel starts a scheduler and executes the initialization program *init*.
- Once *init* has started it can create other processes. In particular, *init* sets the user environment and allows user interaction and the log in, then the kernel keeps itself idle until it is called.
- The process that generates another process is called parent process.
- The process generated by another process is called child process.
- The processes can be parents and children and both at the same time.
- The root of the processes tree is *init*.
- Each process has a unique process ID called PID (Process Identifier).
- The PID of `init` is 1.

2.3 User Interaction

In the past, UNIX mainframe systems had several physical terminals connected to the system via a serial port, often using the RS-232 serial interface (see Figure 2.5). These old terminals often had a green or amber screen and the minimal hardware and logic to send the text typed by the user in the keyboard and display the text received from the mainframe in the screen. In the mainframe, there was a command-line interpreter or *shell*. A *shell* is a process that interprets commands and that provides an interface to the services of the system.

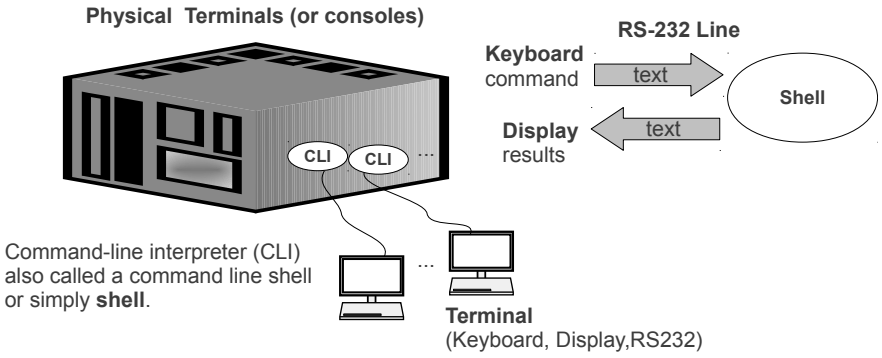
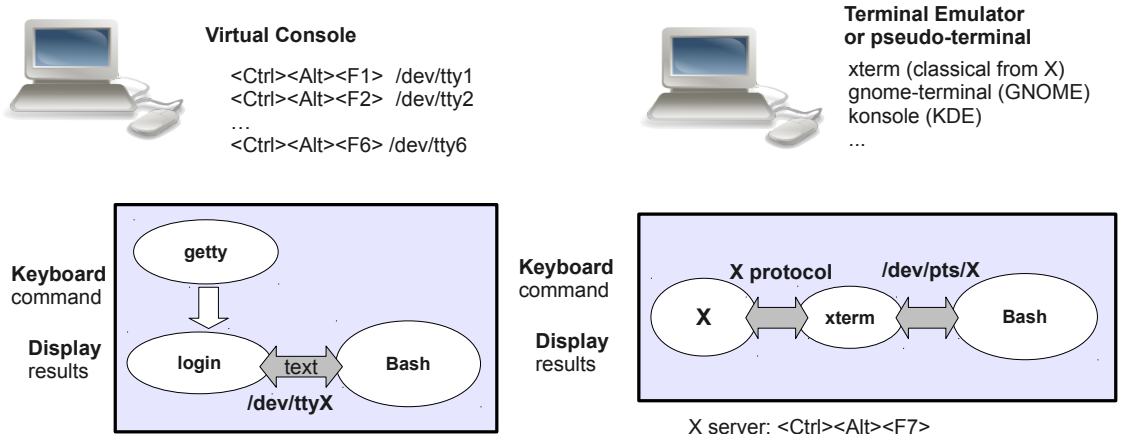


Figure 2.5: Old Physical Terminals.

We must remark that in current systems, we have another interface called Graphical User Interface or GUI. Of course, we still have the CLI (Command Line Interface) interface. In general, CLI gives you more control and flexibility than GUI but GUIs are easier to use. GUI requires a graphical server, called **X server** in UNIX. Processes launched from the GUI are typically applications that have graphical I/O. For this graphical I/O, we can use a mouse, a keyboard, a touch screen, etc. On the other hand, when we exclusively use the CLI, it is not mandatory to have an X server running in the system. Processes launched from the CLI are commands that have only textual I/O. The I/O of

the textual commands is related or "attached" to a terminal and the terminal is also attached to a *shell*. Nowadays we have three types of terminals: Physical terminals, Virtual Consoles and Terminal Emulators. Physical terminals are not very common today, but all the Linux systems implement virtual consoles and terminal emulators (see Figure 2.6).



(a) Virtual Consoles (b) Terminal Emulators

Figure 2.6: Linux Terminals.

If we use virtual consoles to interact with our Linux system, we do not require to start an X server within the system. Virtual consoles just manage text and they can emulate the behavior of several physical terminals. As in general, the user only has a single screen and keyboard, 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**. As shown in Figure 2.6(a), there is a process called `getty` which manages the first stage of the virtual console. Once you log into the system (you enter your user name *login* and *password*), the management is now passed to a process called login, which starts a *shell* (Bash in our case). The communication between the virtual console and the *shell* is not performed using any physical media but using a special file: /dev/ttyX (where X is the number of virtual console).

On the other hand, if our Linux system has an X graphical server running, users can also access the system through a GUI. In fact, GUI is the default interface with the system for most desktop Linux distributions. To access the login manager to the X server, you must type **CRL+ALT+F7** in the majority of the Linux systems. Once you log into the GUI, you can start a terminal emulator (see Figure 2.6(b)). Terminal emulators are also called pseudo-terminals. 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). As you can see in Figure 2.6(b), the terminal emulator communicates with the shell (Bash) also using a file /dev/pts/X (where X is the number of pseudo-terminal). In addition, the pseudo-terminal receives and sends data to the X server using the X protocol.

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 emulator or log into a virtual console, you will see a line of text that ends with the dollar sign "$" and a blinking cursor. Throughout this document, when using "$" will mean that we have a terminal that is open and it is ready to receive commands.

2.4 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.

Chapter 3

Processes

Chapter 4

Processes

4.1 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.

4.2 Working with the terminal

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

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.

Finally, X servers provide a "cut and paste" mechanism. You can easily copy and paste between pseudo-terminals using nothing but your mouse only. Most Linux distros are configured the click of a middle mouse button as paste operation. All you have to do is select text, move your most to another window and hit the middle mouse button. If you have a scroll wheel, just press on it to past the text. If you see only two buttons, just hit both button simultaneously i.e. you can emulate a third, "middle" button by pressing both mouse buttons simultaneously.

4.3 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.
- 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 command `ps` itself. Figure 4.1 shows a scheme of the relationships of all the processes involved when we type a command in a pseudo-terminal.

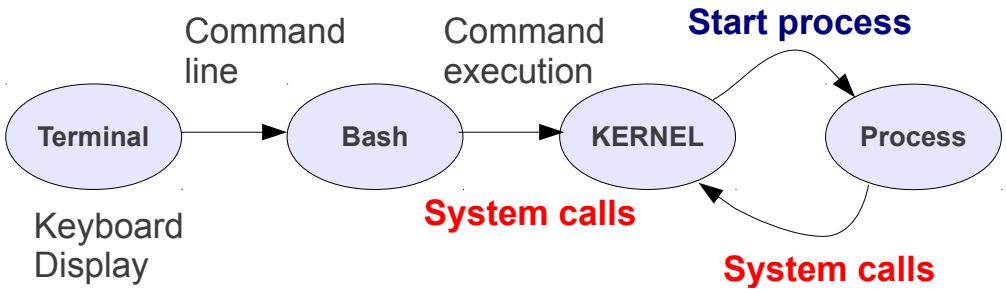


Figure 4.1: Processes related with a pseudo-terminal.

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

```
$ ps -u user1
```

We obtain a list of all the processes owned by `user1`. Next, there is a summary of some relevant parameters of the `ps` command (for further information, you can type `man ps` to view the `ps` manual):

- ***-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.

Example:

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

The preceding command shows the process PID, the PID of parent process, 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.

On the other hand, 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:

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

```
$ 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.

4.4 Scripts

Normally shells are interactive. This means that the shell accepts commands from you (via keyboard) and executes them. But if you use command one by one, then you can store this sequence of commands into a text file and tell the shell to execute this text file instead of entering the commands. This is know as a **shell script**. Another way of defining a shell script is just as a series of command written in plain text file.

Why to write shell scripts?

- Shell script can take input from user, file and output them on screen.
- Useful to create our own commands.
- Save lots of time.
- To automate some task of day today life.
- System Administration part can be also automated.

Example:

```
ps
sleep 2
pstree
```

The previous script command executes a `ps` and then after approximately 2 seconds (sleep makes us wait 2 seconds) a `pstree` command.

To write down a script you can use a text editor (like `gedit`). To run a script you must give it execution permissions (`$ chmod u+x myscript.sh`) and then execute it (`$ ./myscript.sh`).

Another example script is the classical "Hello world" script.

```
#!/bin/bash
# Our first 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 good practice to always include it in scripts. In our case, we will build scripts always for the bash.

As you can see to write to the terminal we can use the `echo` command. 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 myscript.sh
$./myscript.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
echo Please, type a word and hit ENTER
read WORD
echo You typed $WORD
```

Using the first script, notice that the bash clones itself to run the commands within the script.

4.5 Running 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 he can execute a new command. This type of execution is also called **foreground**. For example,

```
$ sleep 5
```

The command simply makes us wait 5 seconds and then terminates. While running a command in foreground we cannot run any other commands.

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:

```
$ sleep 5 &
```

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 in background.

4.6 Signals

A signal is a limited form of inter-process communication. Essentially it is an asynchronous notification sent to a process. A signal is actually an integer and when a signal is sent to a process, the kernel interrupts the process's normal flow of execution and executes the corresponding signal handler. 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. In Linux, the most widely used signals and their corresponding integers are:

- 15 `SIGTERM`. Calls for the process termination.
- 9 `SIGKILL`. Ends the process inmediately.
- 2 `SIGINT`. Is the same signal that occurs when an user in an interactive press **`Control-C`** to request termination.
- 20 `SIGSTOP`. The same signal produced by **`Control-Z`**, stops a process.
- 18 `SIGCONT`. Resumes a previously suspended process by SIGSTSTP.

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 and SIGSTOP are special signals that cannot be captured, that is to say, that are only seen by the kernel. 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.

4.7 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, type:

```
$ 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 &
$
```

In general, typping the JID after the command `bg` will send the process identified by it to *background* (in the previous case the command `bg 2` could be used as well).

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 to terminate (SIGINT) 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.

4.8 Running multiple commands

The commands can be run in some different ways. In general, the command returns 0 if successfully executed and positive values (usually 1) if an error occurred. To see the exit status type `echo $?`. Try:

```
$ ps -ñ
$ echo $?
$ ps
$ echo $?
```

There are also different ways of executing commands:

$ command the command runs in the foreground.

$ command1 & command2 & ... using & one or more commands will run in *background*.

$ command1; command2 ... sequencial running.

$ command1 && command2 && ... command2 is executed if and only if command1 exits successfully (exit status is 0) and so on.

$ command1 || command2 || ... command2 is executed if command1 fails execution. That is to say, if command1 has an exit status > 0 or if command1 has not been executed.

4.9 Extra

4.9.1 *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.

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
```

4.9.2 *Trap: catching signals in scripts

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 CTRL+c.

4.9.3 *The terminal and its 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.

4.9.4 *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*.”

4.9.5 Command Summary

The table 4.1 summarizes the commands used within this section.

Table 4.1: *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.

4.10 Practices

Exercise 4.1– 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 terminal and the command of the currently active processes that have been executed from the terminal. Do the same in the second virtual console.

3. Execute the following commands:

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

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

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 `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. Create a command line using execution of multiple commands that shows the processes launched from terminal but this execution has to be the result of an erroneous execution of a previous `ps` command.

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

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

Exercise 4.2– (*) This exercise deals with additional aspects about processes.

1. 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.

2. 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.

3. Add signal managment to the previous script so that when the USR1 signal is received, the script prints the sentence "waiting operand". Try to send the USR1 signal to the clone Bash executing the script. Tip: to figure out the PID of the proper Bash, initially launch the script in background.

Chapter 5

Filesystem

Chapter 6

Filesystem

6.1 Introduction

File Systems (FS) define how information is stored in data units like hard drives, tapes, dvds, pens, 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, the developer of UNIX). That includes directories. Directories are just files with lists of files inside them. All these 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 "/" but it is not recommended to use the following characters because they have special meaning:

  `= \ ^ ~ ' " ` * ; - ? [ ] ( ) ! & ~ < >`

- The file names are case sentive, that is to say, characters in uppercase and lowercase are distinct.
- 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 images, text, documents, etc. The different FS technologies are implemented in the kernel or in external modules. FS define how the kernel is going to manage the files: which meta-data we are going to use, how the file is going to be accessed for read/write, etc.

Examples of Disk File Systems (DFS) are reiserFS, ext2, ext3, ext4. These are developed within the Unix environment for hard drives, pen drives or UDF (universal disk format) for DVD. In Windows environments we have other DFS like fat16, fat32 and ntfs.

6.2 Basic types of files

Unix kernels manage three basic types of files:

- **Regular files**. These files contain data (as mentioned above).
- **Directory files (folders)**. These files 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 discover the *basic type* 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: -
```

Unix uses the abstraction of "file" for many purposes and thus, 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. Another example of special file is the symbolic link, which we are going to discuss later.

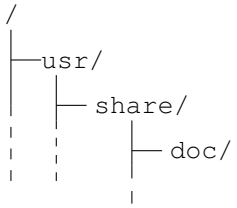
6.3 Hierarchical File Systems

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 are 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	This directory is reserved for system configuration files. This directory cannot contain any binary files (such as programs).
/lib	Contains necessary libraries to run programs in /bin and /sbin.
/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 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).
/home	Contains the working directories of the users of the system except for root.

6.4 The path

As previously mentioned, in Unix-like systems the filesystem has a root denoted as */*. All the files on the system are named taking the root as reference. In general, the *path* defines how to reach a file in the system. 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 root of the filesystem */*.



We have three basic commands to move arround the FS and list its contents:

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

6.5 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 the working directory. If we want to delete it, we can use `rmdir`.

```
$ rmdir ~/myfolder
```

The previous command will fail if the folder is not empty (contains some file or directory). There are two ways to proceed:
(1) Delete the content and then the directory or
(2) Force a recursive removal using `rm -rf`:

```
$ rm -rf myfolder
```

This is analogous to:

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

Where:

- -f: forces to delete.
- -r: deletes the content (recursively).

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 folder contents to other place within the file system the `cp` may be used using the ”-r” modifier (recursive). Example:

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

6.6 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
```

Logically, 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. Moving the file test.txt to the Desktop on the home directory 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 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
```

6.7 File content

Typically some characters appended at the end of the name of files to point out which is the content of a file (these characters are known as the file extension). Examples: text files .txt, jpeg images .jpg or .jpeg, html documents .htm .html etc.

In Unix, the file extension is optional. GNU/Linux uses a guessing mechanism called *magic numbers*, in which some tests are performed to figure out the type of content of the file.

- The command `file` can be used to guess file content of a file.

Example:

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

6.8 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 expansions:

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

Examples:

```
$ cp ~/* /tmp             # Copies all files from home folder to /tmp
$ cp ~/[Hh]ello.c /tmp    # Copies Hello.c and hello.c from home (if they exist)
                          # to /tmp.
$ rm ~/hello?             # Removes files in the home folder called "hello0" or
                          # "hellou" but not "hello" or "hellokitty".
$ rm !(*.jpg)          # Deletes everything 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 single file called hello? but not, for example,
       # a file called hello1.
```

6.9 Links

A link is a special file type which points to another file. Links can be hard or symbolic (soft).

- **A *Hard Link*** is just another name for the same file.
  - The associated name is a simple label stored somewhere within the file system.
  - Hard links can just refer to existent data in the same file system.
  - In most of FS, all files are hard links.
  - Even named differently, the hard link and the original file offer the same functionality.
  - 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 Soft Link)** is considered a new file whose contents are a pointer to another file or directory.
  - If the original file is deleted, the link becomes unusable.
  - The 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 commands create a symbolic link to `/etc/passwd` and print the contents of the link.

```
$ cd
$ cp /etc/passwd .
$ ln passwd hello
$ rm hello
```

This is equivalent to:

```
$ cd
$ cp /etc/passwd hello
```

6.10 Text Files

Text files contain information, usually organized in bytes that can be read using a character encoding table or charset. A text file contains human readable characters such as letters, numbers, punctuation, and also some control characters such as tabs, line breaks, carrier returns, etc. The simplicity of text files allows a large amount of programs read and modify text.

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 7bits. 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 putting 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 or 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, we must take into account the problem of newlines. 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.

Systems based on ASCII or a compatible character set use either LF (Line feed, "`\n`", 0x0A, 10 in decimal) or CR (Carriage return, "`\r`", 0x0D, 13 in decimal) individually, or CR followed by LF (CR+LF, "`\r\n`", 0x0D0A).

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

6.11 Commands and applications for text

Many applications designed to manipulate text files, called text editors, allow the user to save the text content with several encodings. An example is the gnome graphical application "gedit".

On the shell we also have several text editors. A remarkable example is is `vi`[1] or `vim` (a enhanced version of vi). `vi` might be little cryptic text editor, but it is really useful when the basic functions are known. It is present in almost all Unix systems and is necessary whenever the user has not access to a X server. 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, we can navigate through myfile.txt and quit by typing `:q`. If we want to edit the file, we have to press "`i`", which puts `vi` in *insertion mode*. After modifying the document, we can hit `ESC` to go back to *command mode* (default one). To save the file we must type `:wq` and to quit without saving, we 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`. Note: you should not use the previous commands to view executable or binary data files because these may contain non printable characters. The `less` command works in the same way as `man` does. Try:

```
$ cat /etc/passwd
$ 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
```

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

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 `string`, which will find and show characters or groups of characters (strings) contained in a binary file. Try:

```
$ 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
```

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
```

6.12 Unix Filesystem permission system

6.12.1 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 manage internal tasks. For this reason, in addition to real users, in a system there will be other users. Generally, this other users cannot login into the system, i.e., they cannot have a GUI or a CLI.

6.12.2 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. This system controls the ability of the users affected to view or make changes to the contents of the filesystem.

There are three specific permissions on Unix-like systems:

- 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 in order to allow the operating system to run them. When set for a directory, this permission grants the ability to traverse its tree in order to access files or subdirectories, but not see the content of files inside the directory (unless read is set).

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 or directory, the user is in one of the three following categories or classes:

1. Owner Class. The user is the owner of the file or directory.

2. Group Class. The user belongs to the group of the file or directory.

3. 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` for a regular file whose user class has full permissions and whose group and others classes have only the read and execute permissions.

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

`lrw-rw-r--` for a link special file whose user and group classes have the read and write permissions and whose others class has only the read permission.

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
```

6.12.3 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

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 "others" 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
```

Next, we provide some examples in which we illustrate the different ways of using the `chmod` command. For instance, assigning read and execute permissions to the group class of the file temp.txt can be achieved by the following command:

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

Changing the permissions of the file `file1.c` for the user `user1` only to read its contents can be achieved by:

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

For file `file1.c`, we want read permission assignment for all users, write permission only for the owner and execution just for the users within the group. Then, the numeric values are:

r_user+r_group+r_other+w_user+x_group=400+40+4+200+10=654

Thus, the command should be:

```
chmod 654 file1.c
```

6.12.4 Default permissions

Users can also establish the default file 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
```

The two permissions that can be used by default are read and write but not execute. The mask tells us in fact which permission is subtracted (i.e. it is not granted).

6.13 File System Mounting

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 the hard disk 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

   So this, if a CD-ROM or DVD is plugged to IDE bus/connector 1 master device, Linux will show it as `hdc`.

2. Each hard drive has 4 primary partitions (limit of PC x86 architecture).

   - First partition: /dev/hda1
   - Second partition: /dev/hda2
   - Third partition: /dev/hda3
   - Fourth partition: /dev/hda4

3. 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.

Note. You can run command `fdisk -l` to display list of partitions. Example:

```
# fdisk -l /dev/sdb
```

Mounting a Filesystem

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.

If our disk has a second partition, the Kernel will map it in /dev/sda2. Then, we can ”mount“ some part of our filesystem in this second partition. For example, let us consider that we want to store the /home directory in this second partition of sda. The command to mount a storage device under a mount point is `mount`. In the previous example:

```
# 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`.

Unmounting a Filesystem

After using a certain device, we can also ”unmount” it. For example, if the file system of the 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[2]:

```
# 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.

/etc/fstab

Notice that while we can use the command `mount` once the system is running, 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.

6.13.1 Disk usage

Finally, a couple of useful commands are `df` and `du`. The command `df` (abbreviation for disk free) is used to display the amount of available disk space of file systems. `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. Examples:

```
$ df -h
Filesystem            Size  Used Avail Use% Mounted on
/dev/sda1             108G   41G   61G  41% /
none                  1,5G  728K  1,5G   1% /dev
```

[2]Today you can also unmount a mount point using the menus of the GUI.

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

6.14 Extra

6.14.1 *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:

- File type (executable, block special etc)
- Permissions (read, write etc)
- Owner
- Group
- File Size
- File access, change and modification time (remember UNIX or Linux never stores file creation time, this is favorite question asked in UNIX/Linux sys admin job interview)
- File deletion time
- Number of links (soft/hard)
- Extended attribute such as append only or no one can delete file including root user (immutability)
- Access Control List (ACLs)

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 ext3. 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 attribute:

```
$ stat /etc/passwd
```

```
File: '/etc/passwd'
Size: 1988            Blocks: 8          IO Block: 4096   regular file
Device: 341h/833d       Inode: 32820       Links: 1
Access: (0644/-rw-r--r--)  Uid: (    0/    root)   Gid: (    0/    root)
Access: 2005-11-10 01:26:01.000000000 +0530
Modify: 2005-10-27 13:26:56.000000000 +0530
Change: 2005-10-27 13:26:56.000000000 +0530
```

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.

6.14.2 *A file system inside a regular disk file

In this section, we are going to create a file system inside a regular disk file. The example presented in this section is interesting to later understand how the file systems of virtual machines work.
The simplest sequence of commands to create a file system inside a regular disk file is the following:

```
$ dd bs=1M if=/dev/zero of=virtualfs count=30
$ mkfs.ext3 virtualfs -F
$ mkdir vfs
# mount -o loop virtualfs vfs/
```

Next we show in a more step by step fashion, how the previous commands work. In fact, what we use is a loopback device. A more detailed explanation and a more extended set of commands is the following:

- Create a 30MB disk file (zero-filled) called virtualfs in the root (/) directory:

  ```
  dd bs=1M if=/dev/zero of=virtualfs count=30
  ```

- Confirm that the current system is not using any loopback devices.

  ```
  $ losetup /dev/loop0
  ```

  Replace /dev/loop0 with /dev/loop1, /dev/loop2, etc, until a free Linux loopback device is found.

- Attach the loopback device (/dev/loop0) with regular disk file (/virtualfs):

  ```
  $ losetup /dev/loop0 /virtualfs
  ```

- Confirm that the previous step is completed successfully:

  `$ echo $?`

- We are going to create a Linux EXT3 file system on the loopback device that is currently associated with a regular disk file.

  `$ mkfs.ext3 /dev/loop0`

- Create a mount point in /mnt

  `$ mkdir /mnt/vfs`

- Mount the loopback device (regular disk file) to /mnt/vfs as a "regular" Linux ext3 file system.

  `$ mount -t ext3 /dev/loop0 /mnt/vfs`

  Now, all the Linux file system-related commands can be act on this "new" file system.

  For example, you can type `df -h` to confirm its "disk usage".

- To un-mount the loopback file system, type:

  `$ umount /mnt/vfs` related commands.

  and follow with

  `$ losetup -d /dev/loop0`

  to effectively remove the loopback file system and release loopback device subsequently.

6.15 Command summary

The table 6.1 summarizes the commands used within this section.

6.16 Practices

Exercise 6.1– 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.
5. Write two different commands to delete the directories dirA1, dirA2, dirB1 and dirB2 but not dirC1 or dirC2.
6. Delete directories dirC2 and dirC1 using the wildcard "?".
7. Create an empty file in your working directory called `temp`.
8. Type a command for viewing text to display the contents of the file, which obviously must be empty.
9. Type a command to display the file metadata and properties (creation date, modification date, last access date, inode etc.).
10. What kind of content is shown for the `temp`? and what kind basic file is?

Table 6.1: *FS*

stat	shows file metadata.
file	guess file contents.
mount	mounts a file system.
umount	unmounts a file system.
fdisk	show information of a file system.
df	display the amount of available disk space in complete filesystems.
du	file space used under a particular directory or files on a file system.
cd	changes working directory.
ls	lists a directory.
pwd	prints working directory.
mkdir	makes a directory.
rmdir	removes a directory.
rm	removes .
mv	moves a file or folder.
cp	copies a file or folder.
touch	updates temporal stamps and creates files.
ln	creates hard and soft links.
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.
strings	looks for character strings in binary files.
grep	prints lines of a file matching a pattern.
cut	print cuts of a file.
chmod	changes file permissions (or control mask).
umask	shows/sets default control mask for new files.

11. 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?

12. 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`.

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

14. 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 6.2– 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.

2. Open the `vi` text editor and modify the file orig.txt entering some text.

3. Use the command `cat` to view link.txt. What can you observe? why?.

4. Repeat previous two steps but this time modifying first the link.txt file and then viewing the orig.txt file. Discuss the results.

5. Remove all permissions from orig.txt and try to modify the link.txt file. What happened?

6. 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.

7. Delete the file orig.txt and try to display the contents of link.txt with the `cat` command. Then, edit the file with `vi` and enter some text in link.txt. What has happened in each case?

8. Use the command `stat` to see the number of links that orig.txt and link.txt have.

9. 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.

10. Delete the file orig.txt and try to modify with `vi` hard.txt. What happened?

11. 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).

12. Use the `cut` command over the file /etc/group to display the groups of the system and its first five members.

13. 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.

14. 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?

15. Find out what the character is "0x0a", which also appears in the file.

16. 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
```

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

18. 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 7

File Descriptors

Chapter 8

File Descriptors

8.1 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 a 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 expect to have (see Table 8.1).

Table 8.1: *Standard File Descriptors*

Integer value	**Name**
0	Standard Input (stdin)
1	Standard Output (stdout)
2	Standard Error (stderr)

When a command is executed 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    7988 student1    0r   CHR  136,3         5 /dev/pts/3
bash    7988 student1    1w   CHR  136,3         5 /dev/pts/3
bash    7988 student1    2w   CHR  136,3         5 /dev/pts/3
```

As we can see in the column for FD, the file descriptors for 0,1 and 2 are connected to the file /dev/pts/3. The *fd*=0 is opened for reading while *fd*=1,2 are opened for writing. This "mysterious" file /dev/pts/3 is just the file associated with the gnome pseudo-terminal[2].

The "*tty*" file and the file descriptors 0,1 and 2 will be used to send/receive data from/to a textual terminal or pseudo-terminal (see Figure 8.1). We can find the "*tty*" file 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 student1 tty 136, 3 2011-03-03 15:18 /dev/pts/3
```

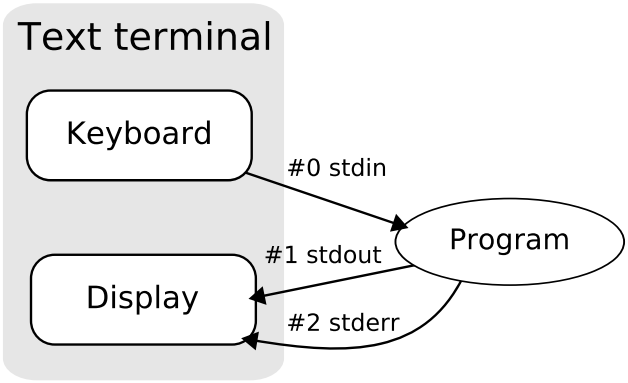


Figure 8.1: Standard Descriptors

Another interesting command is the command `fuser`. Among other functionality, `fuser` shows the PID of the processes that have currently open a certain file. Example:

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

8.2 Redirecting Output

As mentioned, the standard output (stdout) of a process has as default descriptor the number "1". The process inherits this fd from the shell that launched the process. In general, stdout is connected to a special file that represents a terminal.

Similarly, the standard error (stderr) uses the file descriptor number "2". By default, this file descriptor is also connected to the terminal of the shell. The "output redirection" is a feature that allows the Bash to send each of these file descriptors (stdout and stderr) to files different from default ones.

We will show this functionality through a few examples:

[2]When we use the system terminals (accessed with CRL+ALT+F1, etc.) we will find that the files will be in the form: /dev/pts/X where X is the number of pseudo-terminal.

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

As shown, `echo` displays the text that follows the command or in other words it makes an "echo" to standard output (i.e. echoes text to the default terminal).

Now, using $>$ we can redirect standard output (*fd*=1) avoiding the text to appear in the default terminal but we can redirect the output to another file. Let's see an example:

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

On the other hand, using $>>$ we can redirect standard output to a file, but in this case, without deleting the file previous contents but appending the new text at the end of the file. Example:

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

You can also redirect the standard error (*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 standard output or standard error to "*file*". If *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* to add data at the end of the file without erasing its contents.

$\& > file$. Redirects both standard output and standard error to *file*.

Let's consider a more complex example:

```
$ LOGFILE=script.log
$ echo "This sentence is added to $LOGFILE." 1> $LOGFILE
$ echo "This statement is appended to 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.

8.3 Redirecting Input

Basic redirection

Input redirection allows you to specify a file for reading standard input (*fd=0*). Thus, if we redirect the input for a command, the data that the command will use will not come from the terminal (typed by the user) but from the specified file. The format for input redirection is $< file$. We are going to use `myscript.sh` for illustrating the input redirection. Recall that this is script contains the following:

```
#!/bin/bash
# myscript.sh
echo Please, type a word and hit ENTER
read WORD
echo You typed $WORD
```

To redirect the input:

```
$ echo hello > file.txt
$ ./myscript.sh < file.txt
Please, type a word and hit ENTER
echo You typed hello
$
```

As you can observe, the script now reads from file.txt instead of reading from the standard input (the terminal).
We can modify our script to show the table of open file descriptors:

```
#!/bin/bash
# myscript.sh v2.0
# The next line is used to show the open fd (from 0 to 10).
lsof -a -p $$ -d0-10
echo Please, type a word and hit ENTER
read WORD
echo You typed $WORD
```

```
$ ./myscript.sh < file.txt
COMMAND    PID       USER   FD   TYPE DEVICE SIZE/OFF    NODE NAME
myscript 4279 telematics    0r   REG    8,2     6 3277086 /home/telematics/file.txt
myscript 4279 telematics    1u   CHR  136,1    0t0       4 /dev/pts/1
myscript 4279 telematics    2u   CHR  136,1    0t0       4 /dev/pts/1
Please, type a word and hit ENTER
You typed: hello
```

Here Documents

Another input redirection is based on internal documents or " here documents". A here document is essentially a temporary file. The syntax here is: "<<<". The following example creates an here document whose content it the text " hello world" and this temporary file is used as input for the `cat` command.

```
$ cat <<<'hello world'
hello world
```

Finally, to complete the types of input redirection, we discuss the following expression: $<< expr$. This type of construction is also used to create "here documents" in which you can type text until you enter " expr". The following example illustrates the concept:

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

8.4 Unnamed Pipes

Unix based operating systems like Linux offer a unique approach to join two commands on the terminal, with it you can take the output of the first command and use it as input of the second command, this is the concept of pipe or "|". Pipes allow two separate process to communicate with each other also if they were not created to do it, so this open an infinite series of opportunity. A basic example is:

```
$ ls | grep x
```

Bash uses "|" -the pipe symbol- to separate commands and executes these commands connecting the output of a preceding command (`ls` in our example) with the input of the following command (`grep` in our example). In our example, `ls` produces the list of files of the working directory and 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, in this case, the bash shell.

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:

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

The above command line outputs the contents of all the files ending with .txt of the working 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 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 working directory, leaves a copy of these contents in a file called output.txt and then displays these contents in the default 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 the 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.

8.5 Named Pipes

The other sort of pipe is a “named“ pipe, which is sometimes called a FIFO. FIFO stands for ”First In, First Out“ and refers to the property that the order of bytes going in is the same coming out. The ”name“ of a named pipe is actually a file name within the file system.

On older 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 user1  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
```

and in another pseudo-terminal type:

```
t2$ cat < pipe
```

As you see, 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, and 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.

One very useful application of named pipes is to allow totally unrelated programs to communicate with each other. For example, a program that services requests of some sort (print files, access a database) could open the pipe for reading. Then, another process could make a request by opening the pipe and writing a command. That is, the ”server“ can perform a task on behalf of the ”client“. Blocking can also happen if the client is not writing, or the server is not reading.

8.6 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 script depends on the context in which it is used as redirection operator addition, the script has other uses (not discussed in this document). To give an example, the following command: cd “ -”We are commanded to previous working directory (here the script has not been used as operator redirection).

8.7 Process Substitution

When you enclose several commands in parenthesis, the commands are actually run in a "subshell"; that is, the shell clones itself and the clone 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
```

Command 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.

Command substitution also makes the `tee` command (used to view and save the output of a command) much more useful in that you can cause a single stream of input to be read by multiple readers without resorting to temporary files. With command substitution bash does all the work for you. For instance:

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

8.8 Files in Bash

In bash, we can manage a total of 9 file descriptors. 0 is the standard input, 1 is the standard output, 2 is the standard error and there are six 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 output to a file. For this purpose, we could create a script like the following one:

```
#!/bin/bash
LOGFILE=/var/log/script.log
comand1 >$LOGFILE
comand2 >$LOGFILE
...
```

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 use `exec`. Table 8.2 summarizes the functions of exec that will be explained in the below by examples.

Opening in write mode

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

```
#!/bin/bash
LOGFILE=/var/log/script.log
exec 1>$LOGFILE
comand1
comand2
...
```

Table 8.2: *Standard File Descriptors*

Syntax	Meaning
`exec fd> file`	open file for writing and assign `fd`.
`exec fd>>file`	open file for appending and assign `fd`.
`exec fd< file`	open file for reading and assign `fd`.
`exec fd<> file`	open file for reading/writing and assign `fd`.
`exec fd1>&fd2`	open `fd1` for writing. From this moment `fd1` and `fd2` are the same.
`exec fd1<&fd2`	open `fd1` for reading. From this moment `fd1` and `fd2` are the same.
`exec fd>&-`	close `fd`.
`command >&fd`	write stdout to `fd`.
`command 2>&fd`	write stderr to `fd`.
`command <&fd`	read stdin from `fd`.

In the previous example we associate stdout (fd=1) to the log file with `exec` and therefore we no longer have to redirect the output of each command line. Instead of redirecting standard output permanently, it is usual to do it semi-permanently. To do so, you need to backup the original file descriptor (which probably is assigned to a special terminal file /dev/pts/X). The following example is illustrative:

```
$exec 3>&1                # create a new fd that points to
                          # the same place as stdout (fd=1)
$exec 1>script.log        # fd=1 is now redirected to the log file
cmd1                      # the output of cmd1 will go into the log file.
cmd2                      # the output of cmd2 will go into the log file.
exec 1>&3                 # now fd=1 points the same file as fd=3
cat script.log            # Now we see the output in our terminal
lsof -a -p $$ -d0-3       # we have four fd open for this bash
```

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

```
$ exec 3>logfile.log
$ lsof -a -p $$ -d0,1,2,3
COMMAND  PID  USER   FD   TYPE DEVICE SIZE  NODE NAME
bash    3443 student    0u   CHR 136,35         37 /dev/pts/35
bash    3443 student    1u   CHR 136,35         37 /dev/pts/35
bash    3443 student    2u   CHR 136,35         37 /dev/pts/35
bash    3443 student    3u   REG    3,1    0 86956 /home/student/logfile.log
$ echo hello >&3
$ cat logfile.log
hello
$ 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>&-
```

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 read/write mode

Example:

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

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:

```
#!/bin/bash
exec 3>student.log                    # Open fd=3 (for bash)
echo "This goes to student.log" 1>&3  # redirects stdout to student.log
                                      # only for this command line
echo "This goes to stdout"
exec 1>&3                             # redirects stdout to student.log
                                      # for the rest of the commands
echo "This also goes to student.log"
echo "and this sentence, too"
```

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

8.9 Extra

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

8.9.1 Regular Expressions

Introduction

Many commands like `grep` match strings of text in text files using a type of pattern known as a regular expression. Simply stated, a regular expression lets you find strings in text files not only by direct match, but also by extended matches. Regular expressions may be made up of normal characters and/or special characters, sometimes called metacharacters. There are basic and extended regular expressions. Here we will only briefly discuss basic regular and two of the extended regular expressions.

Basic Regular Expression

These have the following meanings:

. 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).

`\( \)` and `\{ \}` are also used in basic RE but they are not going to be discussed here.

Extended Regular Expressions

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.

Some Examples

The following patterns are given as illustrations, along with plain language descriptions of what they match:

`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.

8.9.2 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.

8.9.3 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.

8.9.4 *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[3] arbitrarily long lists of parameters could not be passed to a command. For example, the following command:

```
$ rm $(find /path -type f)
```

May eventually fail with an error message of "Argument list too long", if there are too many files in /path. The same will happen if you type `rm /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
```

The above is equivalent to:

```
$ grep bar $(find . -name "*.foo")
```

However, with xargs 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. However, we must point out that the `find` and `xargs` commands that we use have to support this feature. The next command-line uses -I to tell `xargs` to replace {} with the argument list.

```
$ find . -name "*.foo" -print0 | xargs -0 -I {} mv {} /tmp/trash
```

You may also specify a string after -I that will be replaced. Example:

```
$ find . -name "*.foo" -print0 | xargs -0 -I xxx mv xxx /tmp/trash
```

8.10 Command summary

Table 8.3 summarizes the commands used within this section.

8.11 Practices

Exercise 8.1– 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.

[3] Under the Linux kernel before version 2.6.23

Table 8.3: *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.
`exec fd> file`	open file for writing and assign `fd`.
`exec fd>>file`	open file for appending and assign `fd`.
`exec fd< file`	open file for reading and assign `fd`.
`exec fd<> file`	open file for reading/writing and assign `fd`.
`exec fd1>&fd2`	open `fd1` for writing. From this moment `fd1` and `fd2` are the same.
`exec fd1<&fd2`	open `fd1` for reading. From this moment `fd1` and `fd2` are the same.
`exec fd>&-`	close `fd`.
`command >&fd`	write stdout to `fd`.
`command 2>&fd`	write stderr to `fd`.
`command <&fd`	read stdin from `fd`.
`tr`	translate text.
`find`	search files.
`xargs`	build argument lines.

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 8.2– 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 special file of the pseudo-terminal.

Exercise 8.3– (*) 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 8.4– (*) In this exercise, we will practice with I/O redirection and with Regular Expressions (RE).

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.

Exercise 8.5– (*) In this exercise, we deal with file descriptors are inheritance.

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":

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

Linux Virtualization

Chapter 9

Introduction to Virtualization

Chapter 10

Introduction to Virtualization

10.1 Introduction

Virtualization is a methodology for dividing the resources of a physical computer into multiple operating system (OS) environments. Virtualization techniques create multiple isolated Virtual Machines (VM) or Virtual Environments (VEs) on a single physical server. Virtualized enviroments have three basic elements (see Figure 10.1):

- **Physical Host.** This is the hardware and the OS in which other virtualized machines will run.

  Note. You should not mix up this nomenclature with the term "host" used in networking to refer to a end user computer.

- **Guest or virtual machine**. This is the virtual system running over a physical host. 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.

- **Virtualized network.** The virtual network is composed of a virtual switch that connects the guests like in a real network. As an additional feature, the physical host can provide connectivity for its guests, allowing them to exchange traffic with real networks like Internet.

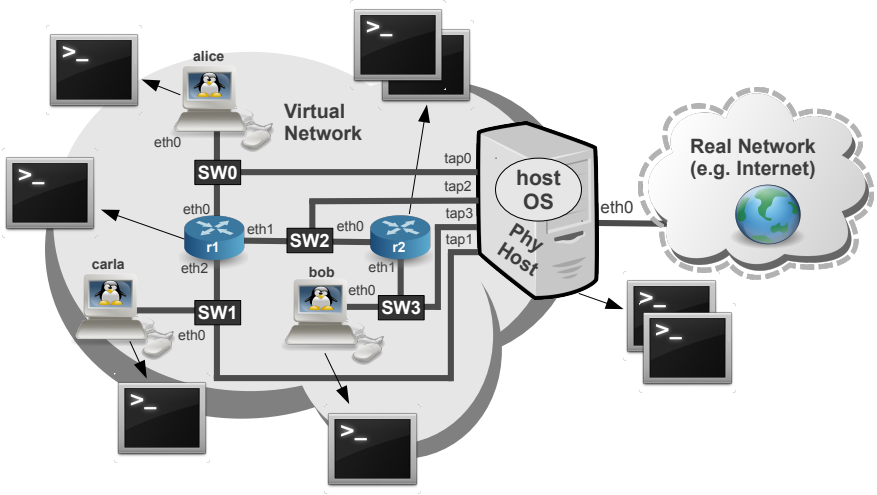


Figure 10.1: Virtualization: a physical host and several guests

10.2 Types of virtualization

There are several kinds of virtualization techniques which provide similar features but differ in the degree of abstraction and the methods used for virtualization.

- **Virtual machines (VMs)**. Virtual machines emulate some real or fictional hardware, which in turn requires real resources from the host (the machine running the VMs). This approach, used by most system emulators, allows the emulator to run an arbitrary guest operating system without modifications because guest OS is not aware that it is not running on real hardware. The main issue with this approach is that some CPU instructions require additional privileges and may not be executed in user space thus requiring a virtual machines monitor (VMM) to analyze executed code and make it safe on-the-fly. Hardware emulation approach is used by VMware products, VirtualBox, QEMU, Parallels and Microsoft Virtual Server.
- **Paravirtualization.** This technique also requires a VMM, but most of its work is performed in the guest OS code, which in turn is modified to support this VMM and avoid unnecessary use of privileged instructions. The paravirtualization technique also enables running different OSs on a single server, but requires them to be ported, i.e. they should «know» they are running under the hypervisor. The paravirtualization approach is used by projects such as Xen, Wine and UML.
- **Virtualization on the OS level, a.k.a. containers virtualization.** Most applications running on a server can easily share a machine with others, if they could be isolated and secured. Further, in most situations, different operating systems are not required on the same server, merely multiple instances of a single operating system. OS-level virtualization systems have been designed to provide the required isolation and security to run multiple applications or copies of the same OS (but different distributions of the OS) on the same server. OpenVZ, Virtuozzo, Linux-VServer, Solaris Zones and FreeBSD Jails are examples of OS-level virtualization.

The three techniques differ in complexity of implementation, breadth of OS support, performance in comparison with standalone server, and level of access to common resources. For example, VMs have wider scope of usage, but poorer performance. Para-VMs have better performance, but can support fewer OSs because one has to modify the original OS. Virtualization on the OS level provides also good performance and scalability compared to VMs. Generally, such systems are the best choice for server consolidation of same OS workloads.

Figure 10.2 shows a picture of the different virtualization types.

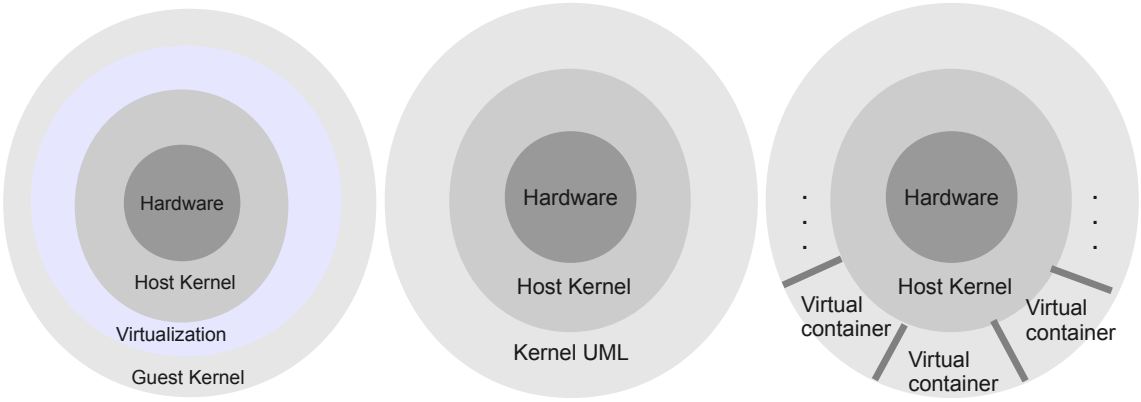


Figure 10.2: Types of Virtualization

10.3 What is UML

We are going to use User Mode Linux (UML). UML was created as a kernel development tool to be able to boot 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 host, just kill the kernel process.

As you can observe in Figure 10.2, UML is a type of Paravirtualization. In particular, UML is designed to be run over another Linux kernel. So UML does not require an intermediate virtualization layer or VMM in the host. Notice that paravirtualization is less complex than VMs but less flexible too: the guest has to be an UML Kernel and host must be a Linux Kernel (but a conventional kernel is enough).

10.4 Practical UML

Let's see how UML works. Let us assume that you have a compiled Linux UML Kernel called "uml-kernel" and also a filesystem within a file called "filesystem.fs" (if you are interested in the details about how to create these things go to section 10.8.1.

Then, to run the UML virtual machine execute the following command:

```
host$ uml-kernel ubda=filesystem.fs mem=128M
```

Note. If the previous command does not work go to Section 10.6.

The previous command executes the UML kernel in the user space of the host. Notice that we have also specified the size of RAM memory that is going to be used. This is a minimal configuration, but UML Kernels support a large number of parameters.

Now, let's see 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. Nevertheless, 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 host file 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 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:

```
host-t1$ uml-kernel ubda=cowfile1,filesystem.fs mem=128M
```

The COW file "cowfile1" need not exist. If it doesn't, the command will create and initialize it. Once the COW file has been initialized, it can be used on its own on the command line:

```
host-t1$ uml-kernel ubda=cowfile1 mem=128M
```

The name of the backing file ("filesystem.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, sometimes you want a new COW file, and you don't want to boot UML in order to get it. This can be done with the `uml_mkcow` command which comes with the uml-utilities package which can be installed in the host system with:

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

In our example:

```
host$ uml_mkcow  cowfile1 filesystem.fs
```

Finally, in another terminal t2 let's fire up our second UML kernel with another COW file (cowfile2):

```
host-t2$ uml-kernel ubda=cowfile2,filesystem.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 host system in any way.

10.5 Update and Install Software in the UML system

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

```
host# mkdir img
host# mount -o loop filesystem.fs img/
host# cp /etc/resolv.conf img/etc/resolv.conf
host# mount -t proc none img/proc
host# chroot img
```

To install software in the system:

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

To finish:

```
host# exit
host# umount img/proc
host# fuser -k img
host# umount img
```

10.6 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:

```
host$ chmod u+x uml-kernel
host$ chmod u+w filesystem.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-kernel, to kill all these processes, we can type the following:

```
host$ killall uml-kernel
```

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

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

Finally, unless otherwise stated, the uml_switch and the UML guests have to be launched with your unprivileged user (not with the root user). If you launch the switch or the UML guest with the root user, you might have problems. You should halt the UML guests, kill the uml_swtich and clean the system as explained above.

10.7 Networking with UML

10.7.1 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-utilites package). An `uml_switch` can be defined as a virtual switch.

UML instances use internally Ethernet interfaces which are connected to the `uml_switch`. This connection uses a Unix domain socket on the host (see Figure 10.3).

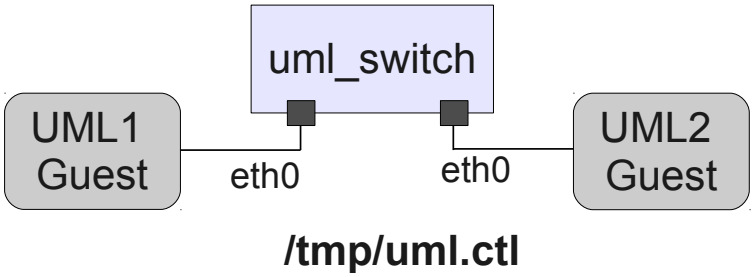


Figure 10.3: Two UML Guests Connected with an `uml_swith`

Command line examples:

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

host-t2$ uml-kernel ubda=cowfile1 mem=128M eth0=daemon

host-t3$ uml-kernel 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
```

10.7.2 Connecting the Host with the UML guests

Now, our goal is to enable network communications between the host and the UML guests (see Figure 10.4).

For this purpose, we need to create a virtual Ethernet network interface in the host and then, connect this virtual interface to the `uml_swith`. The command to create the special network is `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.

```
host# tunctl -u user1 -t tap0
Set 'tap0' persistent and owned by uid 1000
```

The previous command-line creates a special interface called `tap0` and enables *user1* to read/write over `tap0`. Obviously, you must replace "*user1*" with your unprivileged username. 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:

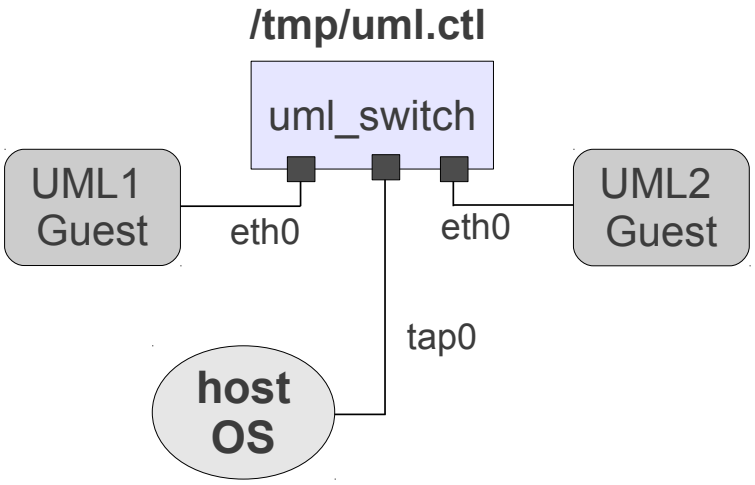


Figure 10.4: Two UML Guests and the Host Connected with an `uml_swith`

```
$ 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 host to an UML guest:

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

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

10.8 Extra

10.8.1 *Building your UML kernel and filesystem

UML Kernel

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

```
host$ mkdir ~/uml
host$ cd ~/uml
```

Copy the Kernel source code to `~/uml`, then untar it and change into the new directory:

```
tar -jxvf linux-XXX.tar.bz2
cd 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 mechanism as a standard Kernel compile, with the exception that every line you type in the process must have the option 'ARCH=um' appended.

To compile make sure that you have the build-essential package installed[1]. To continue, we will create a default configuration to ensure that everything compiles properly.

[1]Or install it with the command `sudo apt-get install build-essential`.

```
~/uml/linux-XXX$ make mrproper ARCH=um
~/uml/linux-XXX$ make defconfig ARCH=um
~/uml/linux-XXX$ make ARCH=um
```

When this completes, you will be have a file 'linux' in the root of the /uml/linux-XXX/ directory. This is your new UML Kernel, optinised to run in user-space. Notice that 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 don't need them so lets remove this debugging info:

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

The UML Kernel that we have contains the default settings and it is compiled to use modules. Once we have created the root filesystem for UML, we will go back to the kernel tree and install the modules into this filesystem, so don't delete the Linux kernel directory just yet.

File system for UML

Next, we will show how to create a basic root file-system that can be launched by the UML kernel to give you a fully functional Linux machine, running inside your normal (host) system, but fully isolated and independent. To create the new virtual file-system, we use the `debootstrap` command. With this command we will create a basic Ubuntu 'natty' system. Firstly, to install `debootstrap`, type:

```
host# apt-get install debootstrap
```

Then, we create a 1GB file to hold the new root file-system. Create the empty file-system and format as ext3:

```
host$ cd ~/uml
host$ dd bs=1M if=/dev/zero of=ubuntu-ext3.fs count=2048
host$ mkfs.ext3 ubuntu-ext3.fs -F
```

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

```
host# mkdir image
host# mount -o loop ubuntu-ext3.fs image/
```

Use `debootstrap` to populate this directory with a basic Ubuntu Linux (natty in this example):

```
host# debootstrap --arch=i386 --include=vim natty image/
```

The program will now contact the Ubuntu archive servers (or a local archive if you specify it as the last option on the command line) and download all the required packages to get a minimal Ubuntu natty system installed. Notice I also asked to install 'vim' since it is my preferred command line text editor. Once it completes, if you list the image directory you will see a familiar Linux root system.

Kernel Modules

Before running this UML, remember that we compiled the Kernel to use loadable modules. We still have to install these modules into the file-system, hence the reason we kept the Linux source available. With the image still mounted, do the following:

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

Modules are stored in the directory `/lib/modules`.

Now, we could actually run this system using the UML kernel and the file-system created, but there are a few little things that we can do first.

Removing Virtual Consoles

One major annoyance if we run the UML now is that 6 separate xterm consoles are opened when it boots. This is due to the fact that Ubuntu by default starts 6 TTY's (accessed by ALT-F1 to ALT-F6 from outside X). If we really only need 1 to open, we can fix this as follows[2]:

1. In the `/etc/event.d/` directory are the startup scripts for tty's. We will delete tty2 to tty6:

   ```
   host# chroot image
   host# cd /etc/event.d/
   host# rm tty2 tty3 tty4 tty5 tty6
   ```

2. In the file /etc/default/console-setup replace:

   `ACTIVE_CONSOLES="/dev/tty[1-6]"` by `ACTIVE_CONSOLES="/dev/tty1"`.

fstab

As a final touch, we will edit the default '/etc/fstab' file to mount the root file-system when the system boots.
Open '/etc/fstab' in your editor and change the contents to the following:

```
/dev/ubda      /      ext3     defaults     0     1
proc           /proc  proc     defaults     0     0
```

Password for root

By default Ubuntu does not have a root password, which makes it impossible to log in. So, we set one:

```
host# passwd
<type your new UML root password here>
<repeat it>
```

Finishing

```
host# exit
host# umount image
```

[2] In old Linux systems virtual consoles are configured in /etc/inittab

Chapter 11

Virtual Network UML (VNUML)

Chapter 12

Virtual Network UML (VNUML)

12.1 Introduction

Defining large topologies is hard and prone to errors. For this reason, it is very helpful to have some systematic way of defining these topologies. To this respect, a research group of the university UPM (Universidad Politecnica de Madrid) of Spain has developed a virtualization tool that allows to easily define and run simulations involving virtual networks using UML. The related project is called VNUML (Virtual Network User Mode Linux). In their web site [1] you can read the following:

> «*VNUML (Virtual Network User Mode Linux) is an open-source general purpose virtualization tool designed to quickly define and test complex network simulation scenarios based on the User Mode Linux (UML) virtualization software. VNUML is a useful tool that can be used to simulate general Linux based network scenarios. It is aimed to help in testing network applications and services over complex testbeds made of several nodes (even tenths) and networks inside one Linux machine, without involving the investment and management complexity needed to create them using real equipment.*»

In short, the VNUML framework is a tool made of two components:

- A **VNUML language** for describing simulations in XML (Extensible Markup Language).
- A **VNUML interpreter** for processing the VNUML language.

Using the VNUML language the user can write a simple text file describing the elements of the VNUML scenario such as virtual machines, virtual switches and the inter-connection topology. Then, the user can use the VNUML interpreter called `vnumlparser.pl` to read the VNUML file and to run/manage the virtual network scenario. This scheme provides also a way of hiding all UML complex details to the user. In the following sections, we provide a description about VNUML and how can we use it.

12.2 Preliminaries: XML

12.2.1 Introduction

The essential component of VNUML is its specification language. As the VNUML language is based on XML, in this section we provide a brief overview of this technology. XML (eXtensible Markup Language) defines a set of rules for encoding documents in a readable form. An XML document is a "text" file, i.e a string of characters coded with UTF8 or with an ISO standard like ISO-8859-1 (Latin1). The characters which make up an XML document are divided into *markup* and *content*. All strings which constitute markup either begin with the character "<" and end with a ">", or begin with the character "&" and end with a ";". Strings which are not markup are content. In particular, a *tag* is a markup construct that begins with "<" and ends with ">". Tags come in three flavors:

[1] http://neweb.dit.upm.es/vnumlwiki

- **start-tags**, for example <section>
- **end-tags**, for example </section>
- **empty-element tags**, for example <line-break />

Another special component in a XML file is the *element*. An *element* is a logical document component that either begins with a start-tag and ends with a matching end-tag or consists only of an empty-element tag. The characters between the start-tag and the end-tag, if any, are the element's content. The element content may also contain markup, including other elements, which are called child elements. An example of an element is `<Greeting>Hello, world.</Greeting>`. A more elaborated example is the following:

```
<person>
  <nif>46117234</nif>
  <name>
    <first>Peter</first>
    <last>Scott</last>
  </name>
</person>
```

Finally, the attribute of an element is a markup construct consisting of a name="value" pair that exists within a start-tag or empty-element tag. For example, the above person record can be modified using attributes to add the age and the gender of the person definition:

```
<person age="17" gender="male">
  <nif>46117234</nif>
  <name>
    <first>Peter</first>
    <last>Scott</last>
  </name>
</person>
```

12.2.2 XML Comments

You can use comments to leave a note or to temporarily edit out a portion of XML code. Although XML is supposed to be self-describing data, you may still come across some instances where an XML comment might be necessary. XML comments have the exact same syntax as HTML comments: they start with "<!--" and end with "-->". Below is an example of a notation comment that should be used when you need to leave a note to yourself or to someone who may be viewing your XML.

```
<person age="17" gender="male">
<!-- Peter is a really nice person -->
  <nif>46117234</nif>
  <name>
    <first>Peter</first>
    <last>Scott</last>
  </name>
</person>
```

12.2.3 Escaping

XML uses several characters in special ways as part of its markup, in particular the less-than symbol (<), the greater-than symbol (>), the double quotation mark ("), the apostrophe ('), and the ampersand (&). But what if you need to use these characters in your content, and you don't want them to be treated as part of the markup by XML processors? For this purpose, XML provides escape facilities for including characters which are problematic to include directly. These escape facilities to reference problematic characters or "entities" are implemented with the ampersand (&) and semicolon (;). There are five predefined entities in XML:

- `&` refers to an ampersand (&)
- `<` refers to a less-than symbol (<)
- `>` refers to a greater-than symbol (>)
- `&apos;` refers to an apostrophe symbol (')
- `&quot;` refers to an quotation symbol (")

For example, suppose that our XML file should contain the following text line:

```
<commnand> echo "1" >/proc/sys/net/ipv4/ip_forward </commnand>
```

The previous line is not correct in XML. To avoid our XML parser being confused with the greater-than character, we have to use:

```
<commnand> echo "1" &gt;/proc/sys/net/ipv4/ip_forward </commnand>
```

In the same way, the quotation mark (") might be problematic if you need to use it inside an attribute. In this case, you have to scape this symbol. Notice however, that escaping the quotation mark is not necessary in our previous example, since the quotation mark appears inside the content of the element (and not in the value of an attribute).

12.2.4 Well-formed XML

A "well-formed" XML document is a text document that satisfies the list of syntax rules provided in the XML specification. The list of syntax rules is fairly lengthy but some key rules are the following:

- The document contains only properly encoded legal Unicode characters.
- None of the special syntax characters such as "<" and "&" appear except when performing their markup-delineation roles.
- The begin, end, and empty-element tags that delimit the elements are correctly nested, with none missing and none overlapping.
- The element tags are case-sensitive; the beginning and end tags must match exactly.
- Tag names cannot contain any of the characters !"#$%&'()*+,/;<=>?@[] \^`{|}~ nor a space character, and cannot start with - (dash), . (point), or a numeric digit.
- There must be a single "root" element that contains all the other elements.

12.2.5 Valid XML

In addition to being well-formed, an XML document has to be "valid". This means that all the elements and attributes used in the XML document must be in the set defined in the language specification and must be used correctly.

For example, if we define a language specification for person registry, we can define the elements: *person*, *nif*, *name*, *first*, *last*. We can also define the person attributes: *age* and *gender* and the type of values for each of the attributes (e.g. *age* attribute is an integer number and *gender* attribute has a value inside the set {male, female}). We might also define the order in which elements can appear and the nesting rules.

For addressing all these issues, XML defines a especial file called "Document Type Definition" (DTD) file. A DTD file defines an XML specification language, including all the elements, attributes and grammatical rules. Finally, the DTD file is used by XML processors to check if an XML document is "valid".

12.3 General Overview of VNUML

12.3.1 VNUML DTD

The VNUML language [2] defines a set of elements, its corresponding attributes and the way these elements have to be used inside an XML document to be a "valid". In Code 12.1 we show the beginning of the DTD file of the VNUML language.

```
<!-- VNUML DTD version 1.8 -->
<!ELEMENT vnuml (global,net*,vm*,host?)>
<!ELEMENT global (version,simulation_name,ssh_version?,ssh_key*,automac?,netconfig?,vm_mgmt?,
          tun_device?,vm_defaults?)>
<!ELEMENT vm_defaults (filesystem?,mem?,kernel?,shell?,basedir?,
          mng_if?,console*,xterm?,route*,forwarding?,user*,filetree*)>
...
```

Code 12.1: Beginning of the VNUML DTD file.

In the previous DTD file we can see several quantifiers. A quantifier in a DTD file is a single character that immediately follows the specified item to which it applies, to restrict the number of successive occurrences of these items at the specified position in the content of the element. The quantifier may be either:

- + for specifying that there must be one or more occurrences of the item. The effective content of each occurrence may be different.
- * for specifying that any number (zero or more) of occurrences are allowed. The item is optional and the effective content of each occurrence may be different.
- ? for specifying that there must not be more than one occurrence. The item is optional.
- If there is no quantifier, the specified item must occur exactly one time at the specified position in the content of the element.

12.3.2 Structure of a VNUML Specification

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE vnuml SYSTEM "/usr/share/xml/vnuml/vnuml.dtd">
<vnuml>
   <!-- Global definitions-->
   <global>
      .......
   </global>
   <!--Network definitions -->
   <net name="Net0" ....   />
   ...
   <!-- Virtual machines definition -->
   <vm name="uml1">
      ...
   </vm>
   ....
</vnuml>
```

Code 12.2: Structure of a VNUML file.

[2] For a extensive description of VNUML language see http://neweb.dit.upm.es/vnumlwiki/index.php/Reference

According to the VNUML DTD file, a VNUML specification document has the structure shown in Code 12.2. The first two lines of the VNUML file, are mandatory for any XML document. The first line is used to check the XML version and the text encoding scheme used. The second line tells the processor where to find the corresponding DTD file.

Following the first two lines, the main body of the virtual network definition is inside the element "vnuml". Inside the <vnuml> tag, we find, in first place, the global definitions section which is marked with the <global> tag. This tag groups all global definitions for the simulation that do not fit inside other, more specific tags. Following the global definitions section, we can find zero or more definitions of virtual networks. These definitions use the tag <net>. Each network created with `<net>` is point of interconnection of virtual machines. This point of interconnection is implemented with a virtual switch like `uml_switch`. The third part of an VNUML specification is devoted to the definition of virtual machines. In this part, we can populate zero or more definitions of UML virtual machines using the `<vm>` tag. The last part of an VNUML specification is devoted to define actions to be performed in the host using the <host> tag. This part is optional and we are not going to use it.

12.3.3 Running an VNUML Scenario

The typical working cycle for running a VNUML scenario is made up of the following phases:

- **Design phase**. The first step is to design the simulation scenario. For this purpose, several aspects have to be considered in advance: the number of virtual machines, the topology (network interfaces in each machine and how they are connected), what processes each virtual machine will execute, etc. Note that the whole simulation runs in the same physical machine (named host). The host may or may not be part of the simulation.

- **Implementation phase**. Once the simulation scenario has been designed, the user must write the source VNUML file describing it. This is an XML file, whose syntax and semantic is described by the Language Reference Documentation of the VNUML project. Also, the XML file must be valid according to the VNUML DTD file.

- **Execution phase**. Once we have written the VNUML file, we can run and manage the scenario executing the `vnumlparser.pl` application.

12.3.4 Simple Example

Before going deeper into the details and possibilities of the VNUML language, we show a simple and auto-explicative example to illustrate the VNUML working cycle. The design phase for this example is described by the topology in Figure 12.1. As shown, the topology is composed of two networks: `Net0` and `Net1`, and three virtual machines: `uml1`, `uml2` and `uml3`, where the machine `uml2` is connected to both networks.

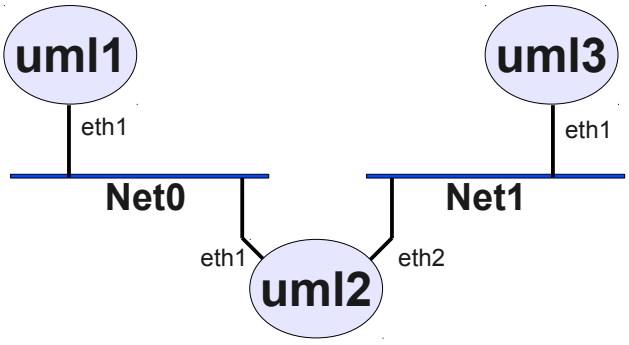


Figure 12.1: Simple Network Topology.

Now, we have to write the VNUML file describing this topology. As mentioned, this is an XML file, whose syntax and semantic is described by the VNUML Language. The Code 12.3 shows a VNUML file that describes the design of Figure 12.1.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE vnuml SYSTEM "/usr/share/xml/vnuml/vnuml.dtd">
<vnuml>
  <!-- Global definitions -->
  <global>
    <version>1.8</version>
    <simulation_name>simple_example</simulation_name>
    <automac/>
    <vm_mgmt type="none" />
    <vm_defaults exec_mode="mconsole">
      <filesystem type="cow">/usr/share/vnuml/filesystems/root_fs_tutorial</filesystem>
      <kernel>/usr/share/vnuml/kernels/linux</kernel>
      <console id="0">xterm</console>
    </vm_defaults>
  </global>
  <!--Network definitions -->
  <net name="Net0" mode="uml_switch" />
  <net name="Net1" mode="uml_switch" />
  <!-- Virtual machines definition -->
  <vm name="uml1">
    <if id="1" net="Net0"></if>
  </vm>
  <vm name="uml2">
    <if id="1" net="Net0"></if>
    <if id="2" net="Net1"></if>
  </vm>
  <vm name="uml3">
    <if id="1" net="Net1"></if>
  </vm>
</vnuml>
```

Code 12.3: VNUML File for a Simple Example

The Code 12.3 is rather auto-explicative but we are going to discuss it a little bit. The **global** section defines that the version of the VNUML language is 1.8. The simulation name is “simple_example”. The MAC addresses of virtual machines are auto-generated. The vm_mgmt tag with the attribute type="none" means that the virtual machines **are not** accessed via a management network shared with the host. Virtual machines are accessed via a console (attribute exec_mode="mconsole"). This console is the `tty0` of the guest and the `xterm` terminal is used in the host to connect to `tty0` in the guest. Virtual machines use COW and all of them use the same root filesystem and kernel. In the **virtual networks** section, we describe two networks in which two `uml_switch` are used to connect the machines of these networks. Finally, in the **virtual machines** section, we describe three virtual machines with names `uml1`, `uml2` and `uml3`. The machine `uml1` has an Ethernet NIC called eth1 which is “connected” to network `Net0`, `uml3` has an Ethernet NIC called eth1 connected to network `Net1`, and `uml2` has two Ethernet NICs called eth1 and eth2, which are connected to networks `Net0` and `Net1` respectively. Now, if we save the previous VNUML description in a file called simple_example.vnuml, we can run the scenario using "-t" option of `vnumlparser.pl`:

```
host$ vnumlparser.pl -t simple_example.xml
```

This command builds the virtual network topology described in simple_example.vnuml and boots all the three virtual machines defined. Once you have finished playing around with the simulation scenario, you can release it using the "-d" option of `vnumlparser.pl`:

```
host$ vnumlparser.pl -d simple_example.xml
```

12.4 VNUML language

In this section, we provide a general description of the VNUML language. For this purpose, we describe the main tags that can be found in each of the three sections of a VNUML specification: the global section, the virtual networks section and the virtual machines section. We will find three kinds of tags in VNUML: structural tags, with few or no semantics; topology tags, used to describe the topology; and simulation tags, used to describe simulation parameters and commands. The following sections describe these tags.

12.4.1 The Global Section

A non-intensive list of tags that may appear inside a <global> tag is the following:

- `<version>`. This tag is required and must be unique.
  Specifies which VNUML language version is being used in the VNUML file. The stable current version is 1.8 and it this the one that we will use.

```
<version>1.8</version>
```

- `<simulation\_name>`. Required and unique.
  Specifies simulation name. Each simulation must have a different name.

```
<simulation_name>simple_example</simulation_name>
```

- `<automac />`. Optional and unique.
  Empty tag. When used, MAC address for the virtual machines interfaces are generated automatically.

```
<automac />
```

- `<vm\_mgmt>`. Optional and unique.
  This tag defines aspects related to a management network interface in the guest to build a management network shared with the host. This defines a way of interacting with the guest from the host. If you do not desire this management network you can use the attribute `type="none"`.

```
<vm_mgmt type="none" />
```

- `<vm\_defaults>`. Optional and unique.
  This tag specifies the values by default that are used by virtual machines. This tag has an attribute called *exec_mode* which is used to indicate the execution mode. One of the "execution modes" is "mconsole", which allows executing commands in the virtual machines from the host without requiring a management network. With the "mconsole" mode, we can specify commands in the VNUML file to be executed when desired (view later the `<exec>` tag). **Note. The UML kernel has to be compiled with support for mconsole to use this functionality.**

```
<vm_defaults exec_mode="mconsole">
```

  In addition, the tags allowed inside this element are the following:

  - `<filesystem>`. Optional and unique.
    It is used to define the "iso" file which will be used as a filesystem of the virtual machine. It accepts an attribute named *type*. When type attribute is set to "COW" (type="cow"), then the `<filesystem>` value is a master filesystem, that will be used in a copy-on-write fashion (COW).
    The way to share a filesystem between two virtual machines is to use the copy-on-write (COW) layering capability of the ubd block driver. This block driver supports layering a read-write private device over a

read-only shared device. A machine's writes are stored in the private device, while reads come from either device. Using this scheme, the majority of data which is unchanged is shared between an arbitrary number of virtual machines, each of which has a much smaller file containing the changes that it has made. With a large number of UMLs booting from a large root filesystem, this leads to a huge disk space saving.
COW mechanism saves a lot of storage space, so COW mode is recommended to boot UMLs. Example:

```
<filesystem type="cow">/usr/share/vnuml/filesystems/root_fs_tutorial</filesystem>
```

- `<mem>`. Optional and unique.
  Specifies the amount of RAM memory used in the virtual machine. Suffixes can be used for (k|K)ilobytes and (m|M)egabytes. The default value is 64M. We can change the default:

```
<mem>128M</mem>
```

- `<kernel>`. Optional and unique.
  Specifies the UML kernel file absolute path name to boot the virtual machine. Note that the file **must be executable**. Example:

```
<kernel>/usr/share/vnuml/kernels/linux</kernel>
```

- `<shell>`. Optional and unique.
  Path to the shell. The default value is /bin/bash. We can change the default:

```
<shell>/bin/sh</shell>
```

- `<basedir>`. Optional and unique.
  Value of the root path used by the `<filetree>` tags (explained later). Example:

```
<basedir>/usr/share/vnuml/scenarios/files</basedir>
```

- `<console>`. Optional and multiple.
  Example:

```
<console id="0">xterm</console>
```

  There are more types of consoles, for more information see the description of the same tag in the section of the virtual machines.

12.4.2 The Section of Virtual Networks

Following the global definitions section, we find the definition of zero or more virtual networks. The `<net>` tag is used for such purpose. The `<net>` tag has some attributes:

- The *name* attribute (mandatory) which identifies the network.
- The *mode* attribute (mandatory) which defines how the interconnection is implemented.
  By default, we will use mode="uml_switch" to indicate that virtual network is in implemented using an `uml_switch` process.

For example, to define a virtual network named Net0 using an `uml_switch` process, we use:

```
<net name="Net0" mode="uml_switch" />
```

There are other attributes that we can find within the `<net>` tag help us to accurately define the behavior of the virtual network. In this context, we can use the *hub* attribute set to "yes" to configure the uml_switch process in hub mode (its default behavior is as switch):

```
<net name="Net0" mode="uml_switch" hub="yes" />
```

Another interesting attribute is the sock attribute, which contains the file name of a UNIX socket on which an uml_switch instance is running. If the file exists and is readable/writable, then instead of starting a new uml_switch process on the host, a symbolic link is created, pointing to the existing socket. In this way, we can create uml_switch instances and set their permissions and their configuration ahead of time. In particular, we can attach a tap interface in the host to the the uml_switch (this can be done using the -tap option of the uml_switch). This allows the host to monitor the virtual networks or to be part of these virtual networks using its tap interface. For example, in the VNUML specification we can use:

```
<net name="Net0" mode="uml_switch" hub="yes" sock="/var/run/vnuml/Net0.ctl" />
```

And then start the uml_switch in the host in the following way:

```
host# uml_switch -tap tap0 -unix /var/run/vnuml/Net0.ctl
```

12.4.3 The Section of Virtual Machines

The virtual machines definition completes the simulation scenario. Virtual machines are defined with the <vm> tag. Each <vm> tag describes a virtual UML machine. The tag uses the *name* attribute to specify the name for the virtual machine. Version 1.8 has a limit of 7 characters for the length of the name.

The optional *order* attribute, that uses a positive integer value (for example, order="2"), establishes the order in which virtual machine will be processed (for example, which virtual machine will be boot/halt first). Virtual machines with no order are processed last, in the same order in which they appear in the VNUML file. You can define as many <vm> tags as you need (including zero). The only restriction is obviously that the names of the virtual machines cannot be duplicated (i.e. the value of the name attribute). Example:

```
<vm name="server">
.....
</vm>
```

Within <vm> several tags configures the virtual machine environment (this is a non-intensive list of tags):

- <filesystem >. Optional (default specified with <vm_defaults>) and unique.
- <mem >. Optional (default specified with <vm_defaults>) and unique.
- <kernel >. Optional (default specified with <vm_defaults>) and unique.
- <console >. Optional and multiple.
  By default, each virtual machine is booted without any I/O channel so, apart networking (supposing it has been configured properly) there is no way for the user to interact with the virtual machine. This approach is fine for some hosts environments (for example, a server where no X server is available) but you may want to have a way of directly accessing to the virtual machine to login and to introduce commands, as you will do in a conventional machine. The <console> comes to solve this problem. It allows you to specify that you want to access the virtual machine through a xterm, a tty line or a pts line. You can specify several consoles (each one with a different id attribute). Examples:
  - If you use the console "xterm", then a tty in the guest is connected to an xterm application in the host. For example:

    ```
    <console id="0">xterm</console>
    ```

    If the previous element is present in the definition of a guest virtual machine, when the simulation is started it will appear an xterm in the host that is connected to the tty0 of the guest.

- If you use the console "pts", then a `tty` in the guest is connected to a pseudo-terminal (pts) in the host. For example:

```
<console id="1">pts</console>
```

If the previous element is present in the definition of a guest virtual machine, then, when the simulation is started, a `pts` in the host is connected to the `tty1` (notice that id="1") on the guest. In particular, the `pts` device is stored by the `vnumlparser.pl` in a file. For example, if your simulation name is "simple_example", and the virtual machine name is "uml1", the filename for the `pts` will be $HOME/.vnuml/simulations/simple_example/vms/uml1/run/pts. If you execute `cat` over this file while the simulation is running, you will obtain a result like this:

```
$ cat $HOME/.vnuml/simulations/simple_example/vms/uml1/run/pts
/dev/pts/7
```

This means that the `/dev/tty1` inside the guest is connected to `/dev/pts/7` inside the host. To access to pseudo-terminal devices, we can use the `screen` command as follows:

```
$ screen /dev/pts/7
```

- `<if >`. Optional and multiple.
This tag describes a network interface in the virtual machine. It uses two attributes: *id* and *net*.
Attribute *id* identifies the interface. The name of a virtual machine interface with id=*n* is `ethn`.
Attribute *net* specifies the virtual network (using name value of the corresponding `<net>`) to which the interface is connected.
Example:

```
<if id="1" net="Net1">
  <ipv4>10.0.1.2/24</ipv4>
</if>
```

As shown in the example, several tags can be used inside <if>:

  - `<mac >`. Optional and unique.
  Specifies MAC address for the interface inside UML. If not used, one address is assigned automatically if `<automac>` is in use or relies in UML.
  - `<ipv4>`. Optional and multiple.
  Specifies an IPv4 address for the interface.
  The mask can be specified as part of the tag value. For example:

```
<ipv4>10.1.1.1/24</ipv4>
```

or using the optional mask attribute either in dotted or slashed notation, for example:

```
<ipv4 mask="/24">10.1.1.1</ipv4>
```

or

```
<ipv4 mask="255.255.255.0">10.1.1.1</ipv4>
```

If the mask is not specified (for example, <ipv4>10.1.1.1</ipv4>) 255.255.255.0 (equivalently /24) is used as default.
Using mask attribute and the mask prefix in the tag value at the same time is not allowed.

  - `<ipv6 >`. Optional and multiple.
  Specifies an IPv6 address for the interface. The mask can be specified as part of the tag value. For example:

```
<ipv6>3ffe::3/64</ipv6>
```

You can also use the optional mask attribute in slashed notation. For example:

```
<ipv6 mask="/64">3ffe::3/64</ipv4>
```

Note that, different from <ipv4> , dotted notation is not allowed in <ipv6>. If the mask is not specified (for example, <ipv6>3ffe::3/64</ipv6>) /64 is used as default.
Using mask attribute and the mask prefix in the tag value at the same time is not allowed.

- <route >. Optional and multiple.
  Specifies a static route that will be configured in the virtual machine routing table at boot time.
  The routes added with this tag are gateway type (gw). Two attributes are used: type (allowed values: "ipv4" for IPv4 routes or "ipv6" for IPv6 routes) and gw, that specifies the gateway address. The value of the tag is the destination (including mask, using the '/' prefix) of the route.

```
<route type="ipv4" gw="10.0.0.3">default</route>
```

- <forwarding >. Optional (default specified with <vm_defaults>) and unique. Activates IP packet forwarding for the virtual machine (packets arriving at one interface can be forward to another, using the information in the routing table). This tag uses the optional *type* attribute (default is "ip"): allowed values are: "ipv4", that enables forwarding for IPv4 only; "ipv6", that enables forwarding for IPv6 only; and "ip" that enables forwarding for both IPv4 and IPv6. The forwarding is enabled setting the appropriate kernel signals under /proc/sys/net.

```
<forwarding type="ip" />
```

Remote command execution: <exec> and <filetree>

VNUML framework has two important features which are very useful to control and manage the virtual machines of a simulation. One of these features is the possibility to perform remote command execution from the host OS (Operating System) to the guests OS when the simulation is running. This feature is accomplished with the **<exec>** tag. Remote command execution offers the possibility to automate procedures over the virtual machines once they are running up.

The other important feature is the remote file copy procedure, which allows to copy files from the host OS to the guest OS's in runtime. This feature is accomplished with the **<filetree>** tag.

<exec>

This is an optional tag and it can appear multiple times in a VNUML file. Specifies one command to be executed by the virtual machine during executing commands sequence mode. In this document, we show the mandatory attributes this tag can use. Optional attributes are described in the VNUML reference manual.

Mandatory attributes:

- *seq*. It is a string that identifies a command sequence. This string is used to identify the commands to be executed.
- *type* (allowed values: "verbatim", "file"). Using "verbatim" specifies that the tag value is the verbatim command to be executed. Using "file", the tag value points (with absolute pathname) to a file (in the host filesystem) with the commands that will be executed (line by line).

In the following example (see Code 12.4), it has been defined two labels as command sequences: "start" and "remove". When the "start" label is executed, the uml1 virtual machine will execute the command "/usr/bin/streamsender" whereas the uml2 virtual machine will execute "/usr/bin/streamreceiver". If "remove" label is executed, then the uml1 virtual machine will execute the command "rm /etc/motd".

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE vnuml SYSTEM "/usr/share/xml/vnuml/vnuml.dtd">

<vnuml>
  <global>
    ....
  </global>
  <net name="Net0" mode="uml_switch" />
  <vm name="uml1">
    ...
    <exec seq="remove" type="verbatim">rm /etc/motd</exec>
    <exec seq="start" type="verbatim">/usr/bin/streamsender</exec>
  </vm>
  <vm name="uml2">
    ....
    <exec seq="start" type="verbatim">/usr/bin/streamreceiver</exec>
  </vm>
</vnuml>
```

Code 12.4: Example exec labels.

<filetree>

This is an optional tag it can appear multiple times in a VNUML file.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE vnuml SYSTEM "/usr/share/xml/vnuml/vnuml.dtd">

<vnuml>
  <global>
    ....
    <vm_defaults exec_mode="mconsole">
      ....
      <basedir>/home/user/config_files/</basedir>
      ...
    </vm_defaults>
  </global>
  <vm name="uml1">
    ...
    <filetree seq="stconf" root="/etc/streamer">streamer/</exec>
  </vm>
</vnuml>
```

Code 12.5: Example filetree labels.

Specifies a filetree (a directory, as well as all its files and subdirectories) in the host filesystem that will be copied to the virtual machine filesystem (overwriting existing files) during execution commands mode. This tag allows easily copying of entire configuration directories (as /etc) that are stored and edited from host when preparing simulations.

- If the directory (in the host filesystem) starts with "/", then it is an absolute directory.
- If the directory doesn't start with "/", then it is relative to **<basedir>** tag.

<basedir> is an optional tag (default specified with <vm_defaults>) and unique. It sets the root path used for <filetree> tags, that is to say, when when the filetree path doesn't start with "/" it uses the path specified in <filetree> as a relative path to the value in <basedir>.
Important note: if <basedir> is not specified, the value of basedir is set to the directory in which it is stored the VNUML file.

<filetree> tag uses two mandatory attributes:

- *root*. Specifies where (in the virtual machine filesystem) to copy the filetree.
- *seq*. The name of the commands sequence that triggers the copy operation. Note that filetree copy is made before processing **<exec>** commands.

Other optional attributes can be viewed in VNUML language reference manual. The code 12.5 shows how to use the **<filetree>** tag. In this example, when label "stconf" is executed, a filetree copy between host and uml1 virtual machine is performed. Specifically, the filetree in the host below "/home/user/config_files/streamer" is copied to the uml1 filesystem at "/etc/streamer".

Note that it is possible to have the same sequence label assigned to an <exec> tag and to a <filetree> tag. In this case, the copy using <filetree> is executed first and next the commands within <exec>.

12.5 The VNUML processor command: `vnumlparser.pl`

Now that we have been introduced to VNUML language and we are able to write an VNUML document for describing a simulation scenario, it's time to present how this VNUML document is executed by the VNUML processor command: "the `vnumlparser.pl`".

As we previously mentioned, the working cycle using VNUML has three phases. The last one, the execution phase, is the phase where we build, start and manage the simulation scenario. This phase is accomplished using an application that processes the VNUML document and executes the appropriate commands: linux, uml_switch, etc. This application for processing the XML file is called in general a "parser"[3]. In VNUML, this parser is called the `vnumlparser.pl`. The `vnumlparser.pl` performs three steps (actually, one of them is optional):

1. **Build scenario.** The parser creates the virtual networks that will interconnect the virtual machines and the host, and then, boots and configures the virtual machines defined, adding IP addresses, static routes or any other network related parameters. The UML boot process makes this step very processor intensive.
2. **Execute commands.** Once the scenario has been built, you can run command sequences on it. Basically in this step, the parser takes the commands defined in the <exec> and <filetree> tags in the VNUML definition file and executes them. Several command sequences may be defined (e.g., one to start routing daemons, another to perform a sanity check in the virtual machines, etc.), specifying which to execute in every moment.
   This step is optional. If you don't need to execute command sequences (because you prefer interact with the virtual machines directly), you don't need it.
3. **Release Scenario.** In this final step, all the simulation components previously created (UMLs, virtual networks, etc.) are cleanly released. The UML shutdown process makes this step also very processor intensive.

`vnumlparser.pl` has several operation modes each related to each of the three steps of the execution phase:

1. Build the scenario: -t mode. The command syntax is:
   `vnumlparser.pl -t VNUML-file`
2. Execute commands: -x mode. In this case once we know the sequence label (*labelname*) we want to execute, the command syntax is:
   `vnumlparser.pl -x labelname@VNUML-file`
3. Release Scenario: -d mode. The command syntax is: `vnumlparser.pl -d VNUML-file`

[3]In computing, a **parser** (syntax analyzer) is one of the components in an interpreter or compiler, which checks for correct syntax and builds data structures related with the input tokens

Chapter 13

Simulation Tools

Chapter 14

Simulation Tools

14.1 Installation

The following instructions allow you to install the tools to build VNUML Virtual Networks and use our `simctl` wrapper. This installation has been tested using the **32-bit version** of Ubuntu 12.04. It is known that the installation **does not work for the 64-bit** versions of these distributions.

In case you have a 64-bit OS, we can provide to you an ISO or VDI (for VirtualBox) image with everything already installed on it.

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. Another possibility, if your physical machine is able to boot from USB (most modern computers support this), is to make a raw copy with `dd` of our ISO image on a USB pendrive and use this device as hard disk.

One you have a Linux box, type the following command to add our repository to your list of APT (software) repositories:

```
$ echo "deb http://sertel.upc.es/~vanetes/debs i386/" |
sudo tee /etc/apt/sources.list.d/simtools.list
```

Finally, type the following commands to update the repository list of software and to install all the packages related to simtools.

Note: you can repeat these steps if the software is not installed correctly at the first time.

```
$ sudo apt-get update
$ sudo apt-get install metasimtools -y --force-yes
```

14.2 A Wrapper for VNUML: `simctl`

With the aim of simplifying and extending the management capabilities of VNUML, the Department of Telematics Engineering (ENTEL) of the UPC has developed several modifications over the `vnumlparser.pl` of VNUML 1.8, a wrapper written for Bash called `simctl` and some other scripts.

The modifications over the `vnumlparser.pl` are essentially for (i) allowing a virtual guest machine to have several consoles connected to several `pts` in the host (mpts functionality) and for (ii) allowing the implementation of virtual networks with other virtual switches like VDE (Virtual Distributed Ethernet).

On the other hand, the script `simctl` allows you to: search for the different scenarios that you can run, start a simulation, stop a simulation, list the virtual machines that are part of a simulation, list the "labels" (seq attributes of <exec> tags) defined on each machine of a simulation, run defined labels, manage the consoles to access the virtual machines, view the network structure, and some more things.

Note. The `simctl` wrapper and other scripts and utilities are distributed as a debian package called simtools. This document describes the version 2.5.15 of simtools.

14.2.1 Profile for `simctl`

The `simctl` wrapper is compatible with the version 1.8 of VNUML but to be able to fully exploit the functionalities of this script you should consider the issues that are listed below:

- We don't use the management network. Thus, we always use:

```
<vm_mgmt type="none" />
```

- The filesystem is always of type COW:

```
<filesystem type="cow">/usr/share/vnuml/filesystems/root_fs_tutorial</filesystem>
```

- We always use the "mconsole" execution mode:

```
<vm_defaults exec_mode="mconsole">
```

- We use consoles of type "pts" and we can use multiple consoles of this type (mpts functionality) as follows:

```
<console id="0">pts</console>
<console id="1">pts</console>
```

  If there are <console> tags in both <vm_defauls> and <vm>, they are merged. Our wrapper, `simctl`, internally uses the `screen` application to automatically manage the connection to these pseudo-terminal devices. Our wrapper is able to list the pseudo-terminals available and to allow you to always connect to the virtual machines. With `simctl`, you will never loose the possibility to have a console with a virtual machine while the simulation is running. You can even close a console and later reopen it without loosing any data. On the other hand, the mpts functionality is implemented by our modified version of the `vnumlparser.pl`, which stores the names of the multiple `pts` devices at the same directory as the original `vnumlparser.pl`, but we use the filenames pts.0, pts.1, etc.

- `simctl` always executes the label "start", i.e. the <exec> tags with attribute seq="start", when it initiates a simulation.

- `simctl` automatically creates a `tap` interface in the host for each virtual network definition in which it founds a sock attribute. For example, if you define a virtual network like this:

```
<net name="Net0" mode="uml_switch" hub="yes" sock="/var/run/vnuml/Net0.ctl" />
```

  Then, `simctl` creates a `tap` interface in the host called `tap0`. In more detail, `simctl` calls another bash script called `simtun`. The script `simtun` is executed with root permissions, which allows us to create the `tap` interfaces and execute the `uml_switch` instances in the host. In this creation, the `tap` is connected with the `uml_switch`, which in turn, is connected with the virtual machines creating the virtual network.

- `simctl` uses a configuration file for defining some basic parameters using the syntax of `bash`. 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`. An example configuration file is the following:

```
# simrc: tunning of environment variables
# for simctl
# Definition of scenario files directory
DIRPRACT=/usr/share/vnuml/scenarios
# Change the default terminal type (xterm)
# values: (gnome | kde | local)
# TERM_TYPE=gnome
# KDE Konsole tunning
# For Konsole version >= 2.3.2 (KDE 4.3.2) use these options:
# KONSOLE_OPTS="-p ColorScheme=GreenOnBlack - -title "
# For Konsole version <= 1.6.6 (KDE 3.5.10) use these options
# KONSOLE_OPTS="- -schema GreenOnBlack -T "
```

The configuration file can be customized. For example, the DIRPRACT environment variable contains the "path" where VNUML simulation files can be found. On the other hand, if you want to use a GNOME terminal instead of an `xterm` terminal, you can assign the variable TERM_TYPE=gnome.

Note. 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.

- With `simctl`, you can always use the "TAB" key to auto-complete the commands and options available at each moment.

14.2.2 Simple Example Continued

Now, following the example of Section 12.3.4, we are going to show how to manage the scenario with `simctl` instead of with the `vnumlparser.pl` and we are going to complete the scenario with more functionality by including the definition of IP network addresses, routes, the execution of commands, multiple consoles etc. Figure 14.1 shows the design of the topology including IP addresses and networks.

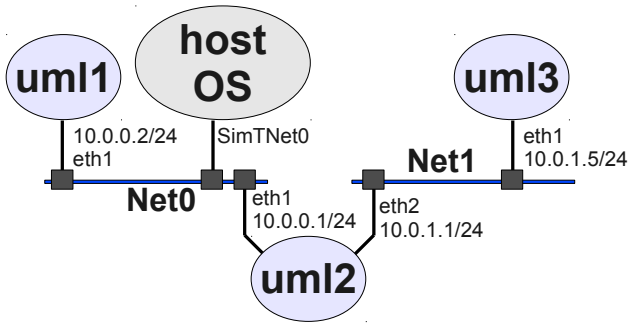


Figure 14.1: Simple Network Topology (Continued).

The Code 14.1 shows a VNUML file that meets the topology and network configuration above exposed. The XML file contains also the configuration of additional functionalities. Next, we discuss the more relevant aspects about this VNUML specification file.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE vnuml SYSTEM "/usr/share/xml/vnuml/vnuml.dtd">
<vnuml>
 <!-- Global definitions -->
 <global>
   <version>1.8</version>
   <simulation_name>simple_example</simulation_name>
   <automac/>
   <vm_mgmt type="none" />
   <vm_defaults exec_mode="mconsole">
     <filesystem type="cow">/usr/share/vnuml/filesystems/root_fs_tutorial</filesystem>
     <kernel>/usr/share/vnuml/kernels/linux</kernel>
     <console id="0">pts</console>
   </vm_defaults>
 </global>
 <!--Network definitions -->
<net name="Net0" mode="uml_switch" hub="yes" sock="/var/run/vnuml/Net0.ctl" />
<net name="Net1" mode="uml_switch" />
 <!-- Virtual machines definition -->
 <vm name="uml1">
   <console id="1">pts</console>
   <if id="1" net="Net0"> <ipv4>10.0.0.2/24</ipv4> </if>
   <route type="ipv4" gw="10.0.0.1">default</route>
   <exec seq="start" type="verbatim">echo "1" &gt;/proc/sys/net/ipv4/conf/all/accept_source_route</exec>
  <exec seq="reset_ips" type="verbatim">ifconfig eth1 0.0.0.0</exec>
 </vm>
 <vm name="uml2">
   <if id="1" net="Net0"> <ipv4>10.0.0.1/24</ipv4> </if>
   <if id="2" net="Net1"> <ipv4>10.0.1.1/24</ipv4> </if>
   <forwarding type="ip" />
   <exec seq="start" type="verbatim">echo "1" &gt;/proc/sys/net/ipv4/conf/all/accept_source_route</exec>
  <exec seq="reset_ips" type="verbatim">ifconfig eth1 0.0.0.0</exec>
  <exec seq="reset_ips" type="verbatim">ifconfig eth2 0.0.0.0</exec>
   <exec seq="enable_forwarding" type="verbatim"> echo "1" &gt;/proc/sys/net/ipv4/ip_forward </exec>
   <exec seq="disable_forwarding" type="verbatim"> echo "0" &gt;/proc/sys/net/ipv4/ip_forward </exec>
 </vm>
 <vm name="uml3">
   <if id="1" net="Net1"> <ipv4 mask="255.255.255.0">10.0.1.5</ipv4> </if>
   <route type="ipv4" gw="10.0.1.1">default</route>
   <exec seq="start" type="verbatim">echo "1" &gt;/proc/sys/net/ipv4/conf/all/accept_source_route</exec>
  <exec seq="reset_ips" type="verbatim">ifconfig eth1 0.0.0.0</exec>
 </vm>
</vnuml>
```

Code 14.1: VNUML File for the Simple Example (Continued)

The first relevant aspect to mention is that the previous definition uses multiple pseudo-terminals. In particular, notice that there is a tag <console id="0"> in the global element of the specification. This means that all the virtual machines will have one console of type "pts". In addition, the definition of virtual machine `uml1` includes another console tag: <console id="1">. This means that this virtual machine is going to have two consoles of type "pts".

Regarding the definition of virtual networks, notice that Net0 has been defined with the sock attribute, which means that this network is going to be connected to the host with a tap interface called `tap0`. The other virutal network defined, Net1, has not the sock attribute, and thus, it will not be connected to any `tap` interface of the host.

Then, we have the definition of the three virtual machines `uml1`, `uml2` and `uml3`. Regarding the IP configuration, as you can observe, we have configured the IP addresses as specified in Figure 14.1 and `uml1` and `uml3` have `uml2` as their default router. The forwarding in `uml2` has been activated too.

Finally, we have several <exec> tags in the definition of each virtual machine. Notice that all the virtual machines have the label “start“ which in this example enables the ”source routing“ functionality of the virtual machines. Again, all the virtual machines have the label ”reset_ips“, which simply removes the IP addresses of the Ethernet network interfaces of the virtual machines (and as a result this action also removes all the routes from the routing tables).

Finally, the virtual machine uml2 has two labels called ”enable_forwarding“ and ”disable_forwarding“ that allow us to enable and disable IP forwarding (which by default is enabled when uml2 is booted).

14.2.3 Getting Started with simctl

Now, if you store the previous file with, for example, the name ”simple_example.vnuml“ in a place in which simctl can locate it (by default, you can use the directory /usr/share/vnuml/scenarios or properly set the variable DIRPRACT in the .simrc file), then you can execute simctl without parameters, and you should obtain the list of possible scenarios that you can run. For example:

```
host$ simctl

simctl ( icmp | routing | subnetting | simple_example ) (OPTIONS)

OPTIONS
   start Starts scenario
   stop Stops scenario
   status State of the selected simulation
                         (running | stopped)
   vms Shows vm's from simulation
   labels [vm] Shows sequence labels for ALL vm's or for vm
   exec label [vm] Exec label in the vms where label is defined
                         or exec the label only in the vm
   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
```

The output of simctl in this example tells us that it has located four scenarios with names: icmp, routing, subnetting and simple_example. This output also shows us all the possibilities that simctl provides us to manage the scenario. These possibilities are explored in the following sections.

14.2.4 Start and Stop Scenarios

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

```
host$ simctl simple_example start
.......
.......
Total time elapsed: 77 seconds
host$
```

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 in the host that the corresponding tap interfaces have been created. In particular, after you run the simple_example scenario, if you type ifconfig -a in the host two view all the network interfaces, you should obtain an output as follows:

```
host$ ifconfig -a
eth1      Link encap:Ethernet  HWaddr 00:23:ae:1c:51:29
          inet addr:192.168.234.252  Bcast:192.168.234.255  Mask:255.255.255.0
          inet6 addr: fe80::223:aeff:fe1c:5129/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:137472 errors:0 dropped:0 overruns:0 frame:0
```

```
          TX packets:105919 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:77824115 (77.8 MB)  TX bytes:10273344 (10.2 MB)
          Interrupt:22 Memory:f6ae0000-f6b00000



tap0      Link encap:Ethernet  HWaddr be:50:6a:c1:55:49
          BROADCAST MULTICAST  MTU:1500  Metric:1
          RX packets:0 errors:0 dropped:0 overruns:0 frame:0
          TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:500
          RX bytes:0 (0.0 B)  TX bytes:0 (0.0 B)

lo        Link encap:Local Loopback
          inet addr:127.0.0.1  Mask:255.0.0.0
          inet6 addr: ::1/128 Scope:Host
          UP LOOPBACK RUNNING  MTU:16436  Metric:1
          RX packets:446 errors:0 dropped:0 overruns:0 frame:0
          TX packets:446 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:0
          RX bytes:16334 (16.3 KB)  TX bytes:16334 (16.3 KB)
```

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

```
host$ simctl simple_example stop
.......
Total time elapsed: 17 seconds
host$
```

Usually stopping a simulation is faster than starting it.

14.2.5 Troubleshooting

If there is a problem starting a simulation you should type the following commands to clear the system and start it again:

```
host$ simctl simulation_name stop
host$ simctl forcestop
host$ simctl simulation_name start
```

The `forcestop` option kills all the ”linux“ processes (UML kernels) and removes the directory .vnuml at the user's home.

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

```
host$ simctl simulation_name1 stop
host$ simctl simulation_name2 stop
host$ 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:

```
host$ sudo -s
host# simctl simulation_name stop
host# simctl forcestop
host# exit
host$ simctl simulation_name start
```

14.2.6 Access to Virtual Machines

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 simple_example scenario already running. At this moment, we can use the command `simctl simple_example vms` to list the virtual machines that are part of the simulation (vms stands for "virtual machines"):

```
host$ simctl simple_example vms
Virtual machines from simple_example:
  1 uml1
  2 uml2
  3 uml3
```

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

```
host$ simctl simple_example get
uml1 Running  --------
uml2 Running  --------
uml3 Running  --------
```

To access to the console of a virtual machine you have to execute "simctl simname get virtual_machine". For example, you can get a command console of the virtual machine *uml1* using the following command:

```
host$ simctl simple_example get uml1
```

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). The argument values (-t) and the terminals can be any of: **xterm** (classic in X11), **gnome** (terminal from GNOME) or **kde** (Konsole from KDE). For example, to get a gnome-terminal for the *uml2* virtual machine, you can type (you can also define this terminal as default in your preferences file "simrc"):

```
host$ simctl simple_example get -t gnome uml2
```

Once you have accessed to the console of the virtual machines *uml1* and *uml2*, you can type:

```
host$ simctl simple_example get
uml1 10121.0 (Attached)
uml2 10169.0 (Attached)
uml3 Running  --------
```

"Attached" indicates that there is already a terminal associated with the command console of *uml1* virtual machine. If you close the *uml1* terminal then the terminal state is "Detached".

Finally, you can also access to the other console of *uml1* using the following command:

```
host$ simctl simple_example get uml1 1
```

Now, if you try the following command:

```
host$ simctl simple_example get
uml1 14272.1 (Attached)
uml1 10121.0 (Attached)
uml2 10169.0 (Attached)
uml3 Running  --------
```

As you observe, *uml1* has two consoles attached. Notice that if you try to get a second console on *uml2* you will obtain an error:

```
host$ simctl simple_example get uml2 1
Error: This virtual machine has not the console 1 enabled
```

14.2.7 Network Topology Information

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

```
host$  simctl simple_example netinfo
UML IFACE NET
uml1 eth1 Net0
uml2 eth1 Net0
uml2 eth2 Net1
uml3 eth1 Net1
```

As you observe, 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.

14.2.8 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. Let us show how the first option ("labels") works with our simple_example scenario.

```
host$ simctl simple_example 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:

```
host$  simctl simple_example 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:

```
host$ simctl simple_example 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
```

Recall that in our example, the "reset_ips" label removes the IP addresses from all the interfaces of the virtual machines. Finally, you can also run a label on a single machine with the following syntax:

```
host$ simctl simple_example exec disable_forwarding uml2
OK The command has been started successfully.
Total time elapsed: 0 seconds
```

Notice that in this particular example, the same result can be obtained executing the previous command without specifying the virtual machine (*uml2*). This is true because in this case, the label "disable_forwarding" is only defined for *uml2*. 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.

14.2.9 Install Software

To install a package XXX in the master filesystem, you have to use the following commands:

```
host$ sudo -s
host# cd /usr/share/vnuml/filesystems
host# mkdir img
host# mount -o loop filesystem.fs img/
host# cp /etc/resolv.conf img/etc/resolv.conf
host# mount -t proc none img/proc
host# chroot img
```

Where "filesystem.fs" must be replaced by the name of the file under the /usr/share/vnuml/filesystems directory that contains the filesystem in which you want to install software. Then, to install software type:

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

To finish:

```
host# exit
host# umount img/proc
host# fuser -k img
host# umount img
host# exit
host$ simctl forcestop
```

14.2.10 Drawbacks of Working with 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 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 ESC. In this mode cursor keys can be used to scroll across terminal screen. To exit scrollback mode press ESC.

Part III

Network Applications

Chapter 15

Introduction to Network Applications

Chapter 16

Introduction

16.1 Introduction

16.1.1 TCP/IP Networking in a Nutshell

In this section, we provide a brief review of the TCP/IP architecture; several more aspects of this architecture will be further discussed in the following chapters.

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 connect a number different networks designed by different vendors into a network of networks (the "Internet"). 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 client and server systems.

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 Figure 16.1):

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

In Unix-like systems, TCP, UDP, IP and ICMP are implemented in the Kernel of the system. These protocols are used to achieve a communication between processes in user space. If a process in user space (application) wants to communicate with another process in the same or another user space, it has to use the appropriate system calls.

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

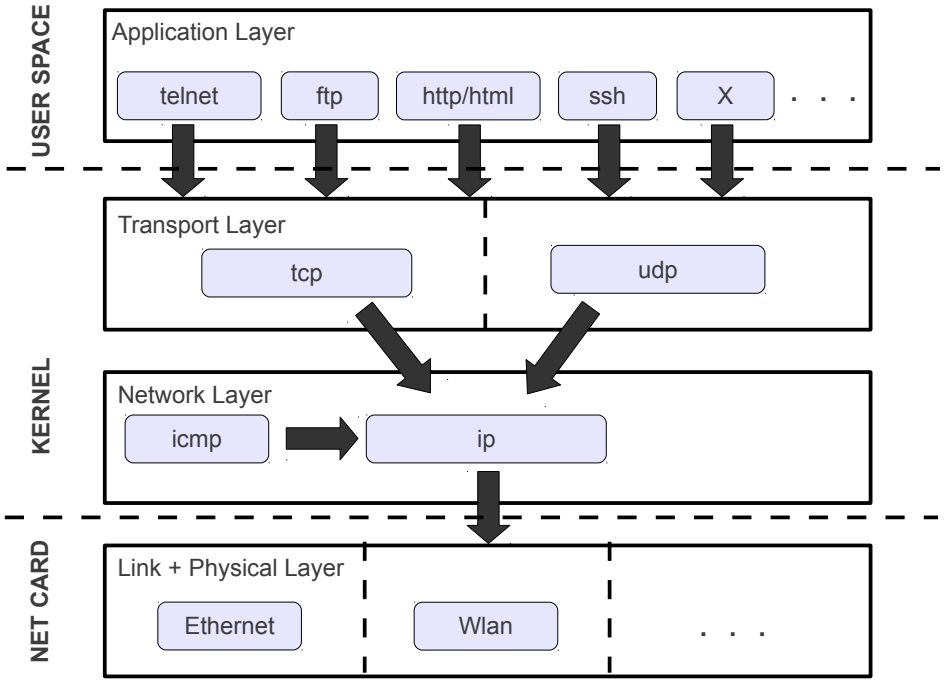


Figure 16.1: Essential TCP/IP protocols and their relationships.

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 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 (see also Section **??**).

Finally, as shown in Figure 16.1, 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 arrows in Figure 16.1 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.

16.1.2 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 16.2). 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.

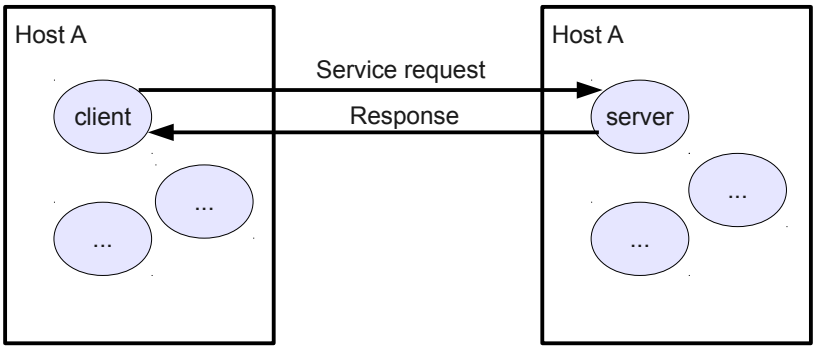


Figure 16.2: Generic client/server model.

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 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 ”socket interface“. A socket is an endpoint of a bidirectional interprocess 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

[1] Further details about this architecture are beyond the scope of this document.

ago. Now, many network services occupy the port number space greater than 1024[2]. For example, the server of the X window system (explained later) 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, at a 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.

16.1.3 TCP/IP 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. To illustrate the previous concepts, we use web browsing as an example. Figure 16.3 shows a summary of the identifiers involved in a TCP/IP communication.

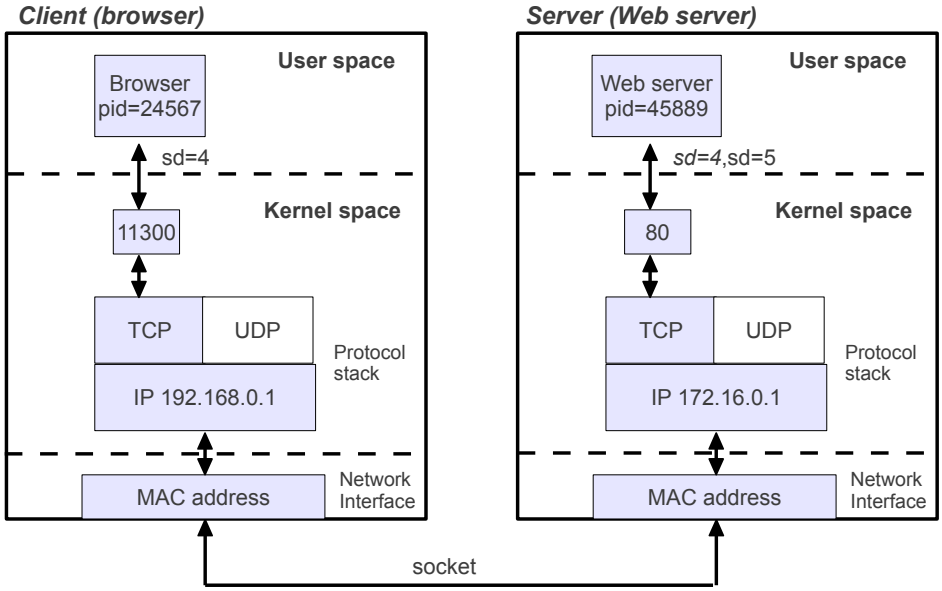


Figure 16.3: Example: WEB browsing with TCP/IP.

So, what happens when you open your favorite browser (e.g. `firefox`) and you type a name in the URL bar like *http://www.upc.edu*? As mentioned, the client (Firefox) needs the tuple {IP address, protocol, port} that identifies the

[2]In most Unix-like systems, only the root user can start processes using ports under 1024.

server process (e.g. `apache` Web server) to establish the socket. The first parameter, the IP address, is obtained from the name[3] that you introduced in the URL bar of the browser. Furthermore, you can also type directly an IP address in this bar, for example, you can type: *http://192.168.0.1*. The second parameter, the transport protocol, is always TCP for HTTP (which is the protocol for the Web service). The third parameter of the tuple, the port, is 80 because this is the default port for this service since this is a well-known service. In the URL bar you can type another port using ":" if the Web server is not running on the the default port, for example, you can type: *http://192.168.0.1:8080*. Then, our client (Firefox) will ask its kernel to establish the socket with the server passing the tuple to the proper system call. Next, the client's kernel will generate a TCP instance and it will negotiate the end-to-end communication with another TCP instance in the server's kernel. After the hand-shake is performed, the client's kernel will provide our WEB browser with a file descriptor (as mentioned, specifically called "socket descriptor"). In addition, when establishing this socket, the client's kernel will assign as an ephemeral transport port for identifying the client process in this TCP/IP end-to-end communication.

In our example of Figure 16.3, our client process (e.g. `firefox` browser) has PID 24567. It has used a system call with the tuple {172.16.0.1,TCP,80} to create a socket with a WEB server. The ephemeral port and the socket descriptor assigned by the client's kernel for this socket are 11300 and 4, respectively. On the other side, we have a Web server (e.g. `apache`) that is running in background with PID=45889, waiting for client connections (or "listening") on port 80. In more detail, the network daemon has bounded server TCP port 80 and it has obtained from its kernel a socket descriptor for this transport port (in our example sd=4). When the WEB server receives an incoming attempt to create a new TCP connection from the remote client, it is notified and it has to use a system call to accept this connection. When a connection is accepted, the server's kernel creates a new socket associated with the corresponding tuple of this connection, which is {192.168.0.1,TCP,11300}, and provides the server process with a socket descriptor (sd=5 in our example).

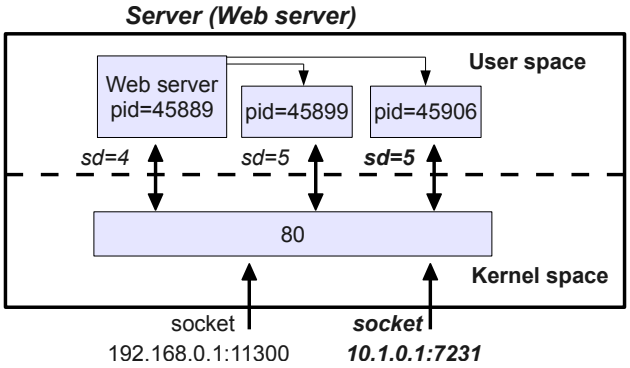


Figure 16.4: Managing several sockets.

On the other hand, most network daemons, and WEB servers are not an exception, use multiple processes[4] to manage multiple connections. In other words, the parent process creates a different child process to serve each connection. This is shown in Figure 16.4. In this example, we have two connections: one from a client in 192.168.0.1 and TCP port 11300 and another connection from a client in 10.1.0.1 and TCP port 7231. Each connection is served by a different process, in this case with PID=45899 and PID=45896. Notice that there is no problem that two or more processes use the same file descriptor (in our example sd=5), because the file descriptor table is local to each process and because the kernel knows which is the TCP/IP tuple associated with the other end of the communication for each socket descriptor. Notice that with this functionality provided by the kernel, a single process can easily manage multiple connections, some as client and some as server simultaneously.

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.

[3]Typically using a DNS service, but you can also use a local file to do this name to IP address translation.
[4]Indeed, multiple execution threads are used.

16.2 Basic Network Configuration

16.2.1 `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 by typing:

```
ifconfig eth0 192.168.0.1 netmask 255.255.255.0
```

16.2.2 `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 16.1, we show some of the most significant parameters of this command.

Table 16.1: *`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.

16.2.3 services

The file /etc/services is used to map port numbers and protocols (tcp/udp) to service names.

16.2.4 `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, network sockets and devices opened by all processes. In this case, we will use `lsof` to list network sockets.

16.3 `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.

16.4 `netcat`

As previously mentioned, with Bash we can open network sockets in client mode but we cannot open server sockets (ports for listening). Fortunately, we have Netcat[5]. This command is a very useful tool for networking because it can be used to open raw TCP and UDP sockets in client and server modes. 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 is as follows (see also Figure 16.5):

- **Send to network.** Netcat sends to the network the data received from stdin. This is performed by the Kernel when Netcat performs a write system call with the corresponding *sd*.
- **Receive from network.** Netcat also writes to stdout the data received from the network. To do so, Netcat uses a read system call with the corresponding *sd*.

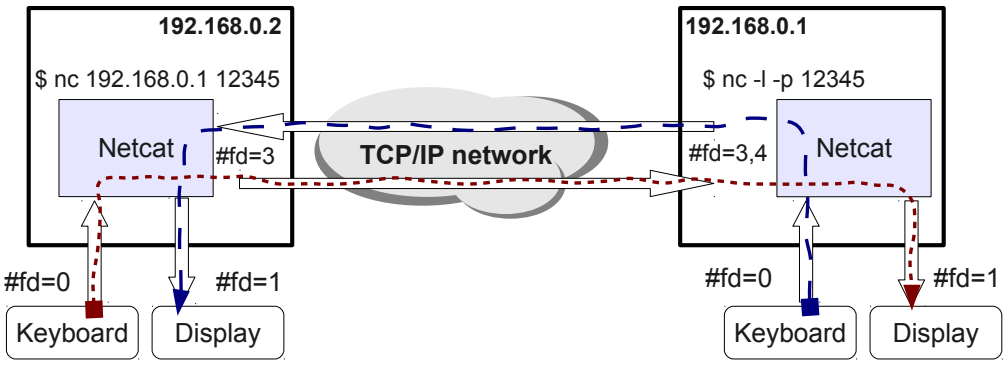


Figure 16.5: File descriptors and `netcat`.

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
```

[5]This command is really useful for management tasks related to network and it is available in most Unix-like platforms, Microsoft Windows, MAC, etc.

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 -tnl | grep 12345
tcp        0      0 0.0.0.0:12345           0.0.0.0:*               LISTEN
```

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 -d0,1,2,3,4,5
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 24.3.

A couple of interesting applications of netcat:

- Transfer files:

  ```
  $ cat file.txt | nc -l -p 12345 -q 0
  ```

- Create a remote terminal:

Table 16.2: *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.

```
$ 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 have to mention that in the Ubuntu distro, the default Netcat application installed is from the netcat-openbsd package. An alternative `netcat` is available in the netcat-traditional package, which is the one that we are going to use. 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
```

16.5 Sockets with Bash

We can open and use client TCP/IP sockets with Bash[6] 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:

[6] In fact, to have this capability, our Bash must be compiled with the –enable-net-redirections flag.

```
echo "Write this to the socket" >&3
```

To read from the socket we can type:

```
cat <&3
```

16.6 Commands summary

Table 16.3 summarizes the commands used within this section.

Table 16.3: *Commands related to network applications.*

`ifconfig`	configures interfaces.
`netstat`	displays system sockets.
`lsof`	lists open files (fd) including sockets.
`ping`	processes echo-request and echo-replay ICMP messages.
`nc or netcat`	opens network sockets.
`firefox`	A WEB browser.
`apache2`	An HTTP server.

Chapter 17

Protocol Analyzer

Chapter 18

Protocol Analyzer

18.1 Introduction to Wireshark

Wireshark has become the "defacto", open-source tool for protocol analysis. It provides low-level packet filtering, analytical capability and it can be used to store captured traffic in a file for later analysis. Wireshark is used by network administrators to troubleshoot network problems, by network security engineers to examine security problems, by developers to debug protocol implementations and in general, people use it to learn network protocol internals. In this section, we just provide a brief introduction to Wireshark, additional features of the protocol analyzer will be explained when required and we also recommend to review the Wireshark User's Guide. So, first, let's install Wireshark. To do so, type:

```
# apt-get install wireshark
```

Once installed, you can run the protocol analyzer from a terminal typing `wireshark`. However, if you run Wireshark with your unprivileged user and you try to click over the "Interface list" (see Figure 18.1), you will get the following error message: "there are no interfaces on which a capture can be done". This is because Unix-like systems are not designed for accessing network interfaces directly but through the socket interface (as we have already seen). Unprivileged users can open sockets (ports above 1024) but they can not capture raw network traffic. For this purpose, you have to be root, so you will need to type (or login as root):

```
$ sudo wireshark
```

After you execute the previous command successfully, you will get the initial Wireshark screen (Figure 18.1).

As you may observe, now you have available the list of all the interfaces of the system[1] for capturing. Then, you can just click on one of the network interfaces to start capturing packets.

18.2 Capturing packets

To capture packets you have to select a network interface. Let us consider that we select the second Ethernet interface "eth2" in our host. When you select an interface, you will be able to see packets moving through that interface.

When there are captured packets available, you can select a packet and view its fields as decoded by wireshark and also in ASCII and hexadecimal. Furthermore, in the bottom left corner of the Wireshark window, the protocol analyzer displays the size of the packet or field selected. For example, in Figure 18.2 we have selected the IP source address of a certain packet. As you can see, Wireshark shows that the IP address occupies 4 bytes as expected.

On the other hand, you can also select the capture options before staring the capture (see Figure 18.3). In our case, we will unset all the "Name resolution" options and we will set always the "Capture packets in Promiscuous mode". This mode allows wireshark to capture data link layer frames that are not destined to our host, i.e. frames that have a

[1] As it is shown in Figure 18.1, recent versions of Wireshark include the list of USB interfaces for capturing but they are not of our interest.

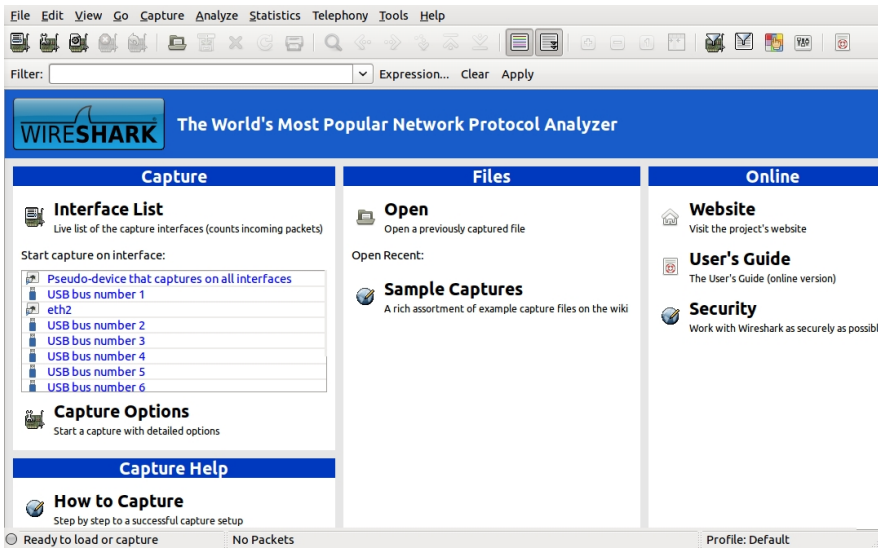


Figure 18.1: Initial Wireshark Screen.

destination link address that is not the one that has our network interface. Finally, notice that you can select a "Capture Filter".

18.3 Capture filter

Capture filters are used to select the data to record in the logs. They are defined before starting the capture. The basic syntax for creating capture filters is the following[2]:

```
(Parameter Value) Logical_Operation (Parameter Value) ...
```

Let's see some examples. For instance, to capture only traffic to or from IP address 172.18.5.4, you can type the following capture filter:

```
host 172.18.5.4
```

To capture traffic to or from a range of IP addresses, you can type the following capture filter (both are equivalent):

```
net 192.168.0.0/24
net 192.168.0.0 mask 255.255.255.0
```

To capture traffic from a range of IP addresses, you can type the following capture filter (both are equivalent):

```
src net 192.168.0.0/24
src net 192.168.0.0 mask 255.255.255.0
```

To capture traffic to a range of IP addresses, you can type the following capture filter (both are equivalent):

```
dst net 192.168.0.0/24
dst net 192.168.0.0 mask 255.255.255.0
```

To capture only HTTP (port 80) traffic, you can type the following capture filter:

[2]The capture filter syntax is the same as the one used by programs using the Lipcap (Linux) or Winpcap (Windows) library like the famous command `tcpdump`.

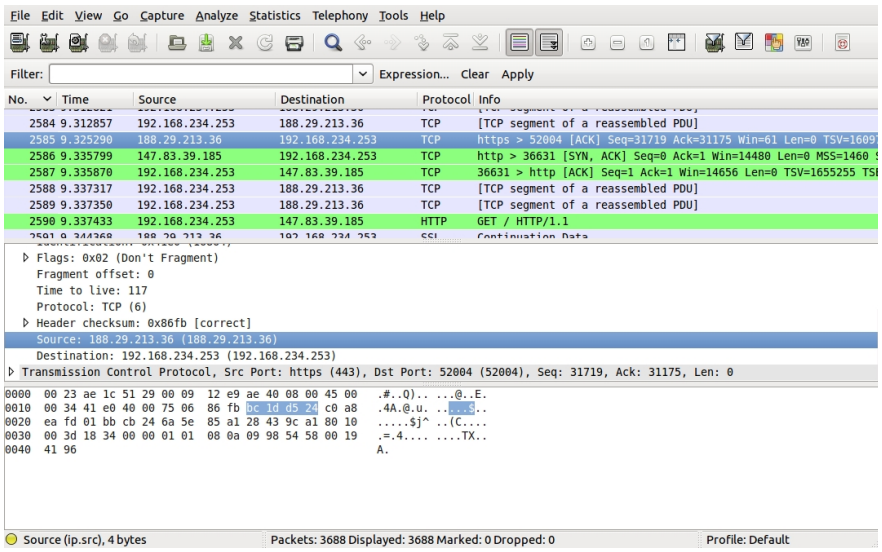


Figure 18.2: Captured Packets.

```
port 80
```

To capture non-HTTP and non-SSH traffic on 192.168.0.1, you can type the following capture filter (both are equivalent):

```
host 192.168.0.1 and not (port 80 or port 22)
host 192.168.0.1 and not port 80 and not port 22
```

To capture all traffic except ICMP and HTTP traffic, you can type the following capture filter:

```
port not 80 and not icmp
```

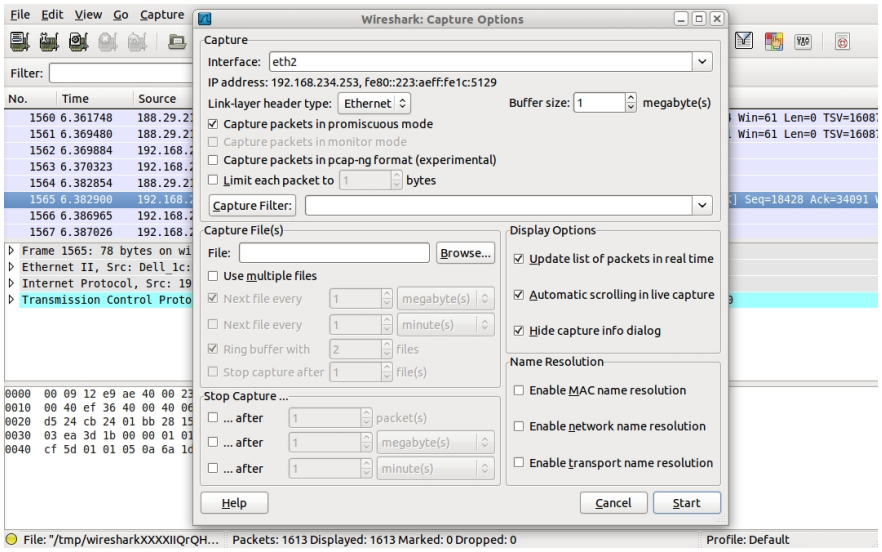


Figure 18.3: Capture Options.

To capture traffic within a range of ports, for example TCP ports between 2001 and 2500, you can type the following capture filter:

```
tcp portrange 2001-2500
```

To capture packets with source IP address 10.4.1.12 or source network 10.6.0.0/16 and having destination TCP port range from 2001 to 2500 and destination IP network 10.0.0.0/8, you can type the following capture filter:

```
(src host 10.4.1.12 or src net 10.6.0.0/16) and tcp dst portrange 2001-2500 and dst
net 10.0.0.0/8
```

18.4 Display filter

Display filters are used to search inside the captured logs. They can be applied and modified while data is being captured. You may wonder if you should use a capture or a display filter. Notice that the goals of the two filters are different. The capture filter is used as a first large filter to limit the size of captured data to avoid generating a log too big. The display filter is much more powerful (and complex); it will permit you to search exactly the data you want.

Let's discuss display filters in more detail. Wireshark uses display filters for general packet filtering while viewing and for its coloring rules. The basics and the syntax of the display filters are described in the Wireshark User's Guide and you can also use the Analyze menu (option Display filters) to build your display filter. Here we just present a few examples. So, for instance, to display only HTTP (port 80) and ICMP traffic, you can type the following display filter:

```
tcp.port eq 80 or icmp
```

To display only traffic between workstations in the LAN 192.168.0.0/16, you can type the following display filter:

```
ip.src==192.168.0.0/16 and ip.dst==192.168.0.0/16
```

To match HTTP requests where the last characters in the URL/URI are the characters "html", you can type the following display filter:

```
http.request.uri matches "html$"
```

Note: The $ character is a regular expression that matches the end of a string, in this case the end of http.request.uri field.

18.5 Follow streams

Another very useful functionality of Wireshark that we are going to use is the "Follow stream". This feature can be selected from the Analyze menu and it works as follows. If you have a TCP or UDP packet selected and you select "Follow TCP stream" or "Follow UDP stream", it will appear in a separate window all the contents of the data stream to which that packet belongs. In addition, the main display of Wireshark will leave the list of packets in a filtered state, with only those packets that are part of that TCP or UDP stream being displayed (see Figure 18.4). You can revert to your old view by pressing ENTER in the display filter text box, thereby invoking your old display filter (or resetting it back to no display filter).

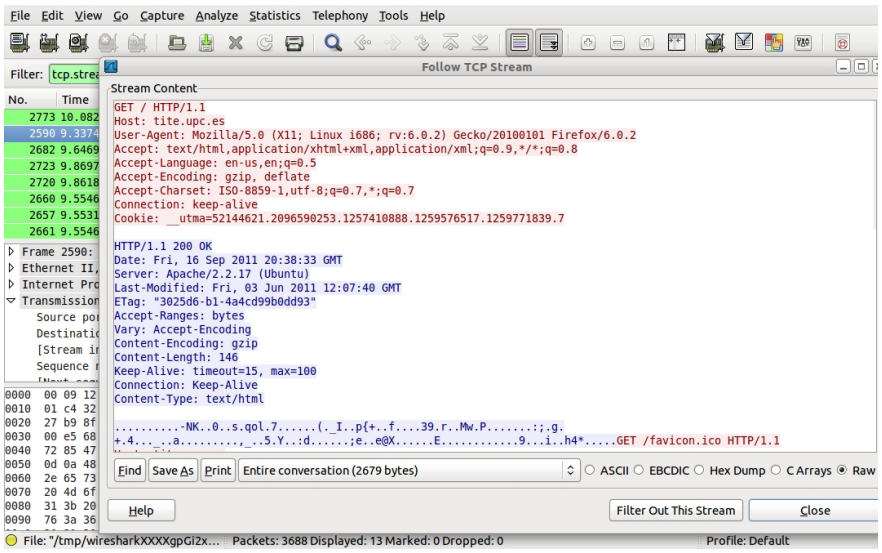


Figure 18.4: Follow TCP stream.

Chapter 19

Basic Network Applications

Chapter 20

Basic Network Applications

20.1 TELecommunication NETwork (TELNET)

20.1.1 What is TELNET?

TELNET or TELecommunication NETwork is a standard Internet protocol for emulating a terminal in a remote host using a TCP/IP network. The TELNET service follows the client/server model. In Linux (and most Unix-like systems) the client is called `telnet` and the server is a daemon called `telnetd`. TELNET is a well-known service over TCP and its default port is 23. 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`.

20.1.2 Practical 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 `telnetd` server in 192.168.0.1:23. The first way is to use the `open` subcommand:

```
$ telnet
telnet>open 192.168.0.1
```

The second way is use a parameters in the command-line:

```
$ telnet 192.168.0.1 23
```

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. If at any moment you need to return to the `telnet` sub-shell, you can to type "CRL+ALTGR+]".

20.1.3 Issues about the 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).

20.2 File Transfer Protocol (FTP)

20.2.1 What is a FTP?

File Transfer Protocol (FTP) is a standard Internet protocol for transmitting files between computers (hosts) on the Internet. FTP uses TCP/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. The control connection remains open for the duration of the session and it is used for session administration (i.e., commands, identification, passwords) 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.

20.2.2 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 TCP connection. In situations where the client is behind a firewall and 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.

20.2.3 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.

20.2.4 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. No End-of-file indicator is needed, unless the data is divided into records.
- Block mode: FTP breaks the data into several blocks (block header, byte count, and data field) and then passes it on to TCP.[5]
- Compressed mode: Data is compressed using a single algorithm (usually run-length encoding).

20.2.5 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 20.1.

Table 20.1: 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.

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:

```
ftp://ftp.upc.edu
ftp://ftpusername@ftp.upc.edu
ftp://ftpusername:password@ftp.upc.edu
```

20.3 Super Servers

20.3.1 What is a super server?

The `inetd` server is sometimes referred to as the Internet "super-server" or "super-daemon" because `inetd` can manage connections for several services. When a connection is received by `inetd`, it determines which program the connection is destined for, spawns the particular process and delegates the socket to it. For services that are not expected to run with high loads, this method uses memory more efficiently, when compared to running each daemon individually in stand-alone mode since the specific servers run only when needed. For protocols that have frequent traffic, such as HTTP, a dedicated or stand-alone server that intercepts the traffic directly may be preferable.

So, when a TCP packet or UDP packet arrives with a particular destination port number, `inetd` launches the appropriate server program to handle the connection. Furthermore, no network code is required in the application-specific daemons, as `inetd` hooks the sockets directly to stdin, stdout and stderr of the spawned process. In other words, the application-specific daemon is invoked with the service socket as its standard input, output and error descriptors.

20.3.2 Configuration

The `inetd` superdaemon 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 launch the program "/usr/sbin/in.telnetd" for the telnet service using 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.

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 result tells us that the telnet service uses the tcp port number 23.

Finally, if you change the configuration file of `inetd`, you have to restart it to apply the changes. To do so, you can use one of the following commands:

```
# /etc/init.d/openbsd-inetd reload
# killall -HUP inetd
```

20.3.3 Notes*

Generally `inetd` handles TCP sockets by spawning a separate application-specific server to handle each connection concurrently. UDP sockets are generally handled by a single application-specific server instance that handles all packets on that port. Finally, some simple services, such as ECHO, are handled directly by `inetd`, without spawning an external application-specific server.

You can also run a telnet daemon standalone. For this purpose, you can use `in.telnetd` in debug mode typing:

```
in.telnetd -debug <port number> &
```

When a daemon is started stand-alone it must open the port for listening, while when the daemon is started through `inetd`, it just uses `stdin`, `stdout` and `stderr`, which are inherited from `inetd`.

20.3.4 Replacements*

In recent years, there have appeared several new super-servers that can be used to replace `inetd`. One of the most extended is `xinetd` (eXtended InterNET Daemon). The `xinetd` super-daemon offers a more secure extension to or version of `inetd`. This super-server features access control mechanisms, extensive logging capabilities, and the ability to make services available based on time. It can place limits on the number of servers that the system can start, and has deployable defense mechanisms to protect against port scanners, among other things. The global configuration of `xinetd` can be usually found in the file /etc/xinetd.conf, while each application-specific configuration file can be found in the /etc/xinetd.d/ directory.

20.4 Commands summary

Table 20.2 summarizes the commands used within this section.

Table 20.2: *Commands related to basic network applications.*

`telnet`	TELNET protocol client.
`telnetd`	TELNET protocol server.
`inetd and xinetd`	Super servers.
`ftp`	FTP protocol client.
`ftpd`	FTP protocol server.

20.5 Practices

20.6 Practices

Exercise 20.1– Start the scenario *basic-netapps* on your host platform by typing the following command:

```
host$ simctl basic-netapps start
```

This scenario has two virtual machines `virt1` and `virt2`. Each virtual machine has two consoles (0 and 1) enabled. E.g. to get the console 0 of `virt1` type:

```
host$ simctl basic-netapps get virt1 0
```

1. Open the console 0 in `virt1`, which we call *virt1.0* and also open *virt2.0*.
2. Figure out which is the port number of the service *daytime* using the configuration file `/etc/services` of `virt1`.
3. In a windows-like OS, what port number you expect for the service *daytime*?
4. List the services that are active in `virt1` under `inetd`.
5. Annotate the MAC and IP addresses of each interface on `virt1` and `virt2`.
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. Assign the IP addresses 192.168.0.1 and 192.168.0.2 to the ethernet interfaces of `virt1` to `virt2`, respectively.
8. Send 3 ICMP echo-requests from `virt2` to `virt1` with the `ping` command.
9. Restore the original IP addresses of each ethernet interface of each machine.

Exercise 20.2– Using the scenario *basic-netapps*, we are going to analyze the TELNET service.

1. Start a capture of `tap0` with `wireshark` in the host platform.
2. You can check that you can do a telnet from `virt1` to `virt2` but not on the other way. So, activate the telnet service under `inetd` in `virt1`. Note. Be careful to not to leave any space at the beginning of `inetd` configuration lines.
3. Establish a telnet session using the `root` user from `virt2` to `virt1`. Use the console *virt2.0*. Open the console 1 in `virt2`. In each console of `virt2`, use the `netstat` command with the appropriate parameters to check the ports used in 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" under the Analyze menu 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` with the command:

   ```
   virt2.0# ssh 10.1.1.1
   ```

   Use the follow TCP stream option of `wireshark` and comment the differences between SSH with TELNET about ports used and security.

Exercise 20.3– Using the scenario *basic-netapps*, we are going to use the Web.

1. Start the `apache2` Web server on `virt1` using the console *virt1.0*. To do so type the following command:

   ```
   virt1.0# /etc/init.d/apache2 start
   ```

   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. Edit the file /var/www/index.html with `vi` in `virt1` and change some of the text (respecting the HTML tags). Using a console in `virt2` and the `lynx` command, which is a console (text-based) web browser, display the web page of `virt1`.
3. Start a new capture of `tap0` with `wireshark` in the host platform. Give the IP address 10.1.1.3 to the `tap0` interface and open a `firefox` to see the web page served in `virt1`. Use the follow TCP stream option of `wireshark` and roughly comment the protocols that you see.

Exercise 20.4– Using the scenario *basic-netapps*, we are going to use and analyze the tool `netcat`.

1. Start a new capture of `tap0` with `wireshark` in the host platform and try the following command:

   ```
   virt1.0$ nc -l -p 12345
   ```

Describe what the previous command does and also check the ports and open files related to the `netcat` process.

2. Now try:

   ```
   virt2.0$ nc 10.1.1.2 12345
   ```

   Describe the ports, the open files of each `netcat` process. Send some text from each machine and terminate the connection. Describe also the capture and the behavior of the previous commands.

3. Create and test a command line with `netcat` that transfers the file /etc/services. Use the port 23456. Run the command on `virt1`. From `virt2` try to connect to the service with the commands `nc` and `telnet`.

4. Create and test a command line with `netcat` that emulates the daytime service using the port 12345 (**hint:** use the `date` command). Run the command on `virt1`. From `virt2` try to connect to the service with `nc`.

Exercise 20.5– Using the scenario *basic-netapps*, we are going to create a service under `inetd`.

1. Create a small script with `netcat` that listening on port 5555 gives us the amount of free disk of the host (**hint:** use the `df` command). Explain the configuration, your tests and explain in as much detail as possible the behavior of your network service: ports used, relationship between processes, filedescriptors, general behavior of the service etc.

2. Implement the same service using `inetd` (without using `netcat`). Explain the configuration in the system, the differences with respect to the `netcat` implementation and explain all details possible too.

Exercise 20.6– Using the scenario *basic-netapps*, we are going to analyze the FTP service.

1. You must allow the ftp access for the root user in `virt1`. To do so, 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.

2. Start a new capture of `tap0` with wireshark in the host platform. Then, establish an FTP session from `virt2` to `virt1` using the root user and the console *virt2.0*. Use the follow TCP stream option of `wireshark` and comment the protocol.

3. Using the console *virt1.0* allow the ftp access to the root user. This is done by commenting the line that contains root in the file /etc/ftpusers.

4. Start a new capture of `tap0` with `wireshark` in the host platform. Establish an FTP session from `virt2` to `virt1` using the root user and 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`.

5. Get all the files in /usr/bin that start with "z" and exit. Which is the default data representation for these transmissions?

6. Use the follow TCP stream option of `wireshark` to comment the FTP dialogue that you see with port 21 of the server.

7. Figure out also how the data (files) are transferred and which ports are used.

8. 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.

9. (*) Now we are going to see the differences between the binary and ascii data representations. Download the files /tmp/text-lf.txt and /tmp/text-crlf.txt in binary mode (default). In the client, rename them respectively as text-cr-binary.txt and text-crlf-binary.txt.

10. (*) Now, download the same files in ascii mode. In the client, rename them respectively as text-lf-ascii.txt and text2-crlf-ascii.txt.

11. (*) Use the commands `hexdump` and `file` in the client and the server to analyze these files. Explain your conclusions and explain if our ftp client changes the line endings of the original files.

12. (*) Analyze with `wireshark` the number of bytes sent in each transmission. When transmitting the file text-lf.txt, why is 11 bytes in binary mode and 13 bytes in ascii mode?

13. (*) Finally, try the passive mode and transfer any file you like. Explain how you do it and analyze and discuss the differences regarding to active mode.

Part IV

Linux Advanced

Chapter 21

Shell Scripts

Chapter 22

Shell Scripts

22.1 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 4.4. This section, by means of examples, makes a more detailed introduction to the possibilities that scripts offer.

22.2 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:

```
#!/bin/bash
MYVAR='Hello world'
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:

```
#!/bin/bash
MYVAR='Hello world'
MYVAR2= '$MYVAR, How are you?'
echo $MYVAR2
```

[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 22.1: 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*.

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 (to be able to assign a value with spaces) you have to use double quotes. In general, there are three types of quotes as shown in Table 22.1. Now, if you try the following script you will obtain the desired result:

```
#!/bin/bash
MYVAR='Hello world'
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}.

22.3 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:

```
#!/bin/bash
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.

22.4 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

22.4.1 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
```

22.4.2 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
```

22.4.3 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 22.8.4.

22.4.4 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
```

22.4.5 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`.

22.4.6 Process Substitution

Process substitution is explained in Section 8.7.

22.4.7 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 22.2: 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.

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
```

22.5 Conditional statements

In this section we present conditional statements available for shell scripts.

22.5.1 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 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).

22.5.2 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 22.3).

Table 22.3: 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.

22.5.3 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).

22.5.4 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:

```
#!/bin/bash
X=0
while [ $X -le 20 ]
do
      echo $X
      X=$((X+1))
done
```

The loop is executed while the variable `X` is less or equal (-le) than 20. We can also create infinite loops, example:

```
#!/bin/bash
while true
 do
  sleep 5
  echo "Hello I waked up"
 done
```

22.5.5 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:

```
case VARIABLE in
    pattern1)
```

```
        1st block of code ;;
    pattern2)
        2nd block of code ;;
    ...
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:

```
#!/bin/bash
for FILE in $*; do
  case $FILE in
   *.jpg | *.jpeg | *.JPG | *.JPEG)
   echo The file $FILE seems a JPG file.
   ;;
  *.avi | *.AVI)
   echo "The filename $FILE has an AVI extension"
   ;;
  -h)
   echo "Use as: $0 [list of filenames]"
   echo "Type $0 -h for help" ;;
  *)
   echo "Using the extension, I don't now which type of file is $FILE."
   echo "Use as: $0 [list of filenames]"
   echo "Type $0 -h for help" ;;
  esac
done
```

The final pattern is *, which is a catchall for whatever didn't match the other cases.

22.6 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
```

22.7 Functions and variables

22.7.1 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
}
ver_zip $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.

22.7.2 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 `let` or equivalently "`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 shell variables have one of the following scopes: local variables, environment variables, position parameters and special variables.

Local Variables

Simply stated, local variables are those variables used within a script, 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=hello

my_function(){
VAR="one two three"
for X in $VAR
  do
  echo -n $X
  done
}

my_function
echo $VAR
```

When you run this script the output is: 'onetwothreethree". 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=hello

my_function(){
local VAR="one two three"
for X in $VAR
  do
  echo -n $X
  done
}

my_function
echo $VAR
```

In this case, the output is: 'onetwothreehello', so the global variable VAR is not affected by the execution of the function.

Environment variables

Any process in the system has a set variables that defines its execution context. These variables are called *environment or global variables*. Shells are not an exception. Each shell instance has its own environment variables. If you open a terminal, you can add environment variables to the associated shell in the same manner you define variables in your scripts. Example:

```
$ ps
  PID TTY          TIME CMD
```

```
 5785 pts/2    00:00:00 bash
 5937 pts/2    00:00:00 ps
$ MY_VAR=hello
$ echo $MY_VAR
hello
```

In the previous example, we added MY_VAR as a new environment variable for the bash instance identified by PID 5785. However, if you create a script that uses MY_VAR and execute it from the previous terminal, you will see that the variable is not declared. This is because when you execute a shellscript from a terminal or a pseudo-terminal, by default, the associated bash creates a child bash, which is who really executes the commands of the script. For this execution, the child bash only inherits the **"exported context"**. The problem is that variables (or functions) are not exported by default. To export a variable (or a function) you have to use the keyword `export` or alternatively `declare -x`:

- **Export a variable**. Syntax: `export variable` or `declare -x variable`.
- **Export a function**. Syntax: `export -f function_name`.
- **View context**. Syntax: `declare`.
- **View exported context**. Syntax: `export` or `declare -x`.

For example, create a script called env-script.sh as follows:

```
#!/bin/bash
# file: env-script.sh
echo $MY_VAR
```

Then, execute the following command-lines:

```
$ MY_VAR=hello
$ echo $MY_VAR
hello
$ declare
...
MY_VAR=hello
...
$ ./env-script.sh
$ export MY_VAR
$ export
...
MY_VAR=hello
...
$ ./env-script.sh
hello
```

Notice that env-script.sh can use the variable 'MY_VAR' only after it is exported. In general, exported variables and functions get passed on to child processes, but not-exported variables do not. 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 also loaded. There are several configuration files in which the system administrator or the user can set the initial environment variables for bash like /.bashrc, /etc/profile etc. Nevertheless, a detailed description of this issue is out of the scope of this document.

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 22.4. Recall that the values of the environment variables `HOME` and `PWD` can also be obtained using tilde expansions: ~ and ~+.

Table 22.4: 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.

The command `source`

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:

```
#!/bin/bash
echo PID of our parent process $PPID
echo PID of our process $$
echo Showing the process tree:
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 from PID=2119:
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 22.7.1, 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:

```
#!/bin/bash
# file my_zip_lib.sh

OPTION="-l"
VAR="another variable..."

zip_contents() {
 echo "Contents of $1: "
  unzip $OPTION $1
}
```

Now, we can use '`source`' to make the function and the variables defined in this file available to our script.

```
#!/bin/bash
# script.sh

source my_zip_lib.sh

echo The variable OPTION from sourced from my_zip_lib.sh is: $OPTION
echo Introduce the name of the zip file:
read FILE
zip_contents $FILE
}
```

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 parameters

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 22.3. 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.

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 $$ contains the PID of the running process. The variable $$ 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 22.5 briefly summarizes these variables.

22.8 Extra

22.8.1 *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:

Table 22.5: 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.

```
#!/bin/bash
echo "Hello $USER,"
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.

22.8.2 *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:

```
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"
```

22.8.3 *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 22.6 shows bash builtins.

22.8.4 *More on parameter expansions

Simple usage

```
$PARAMETER
${PARAMETER}
```

Table 22.6: *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

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
```

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/XX}
toXX is my day
$ echo ${VAR//day/XX}
toXX is my XX
```

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.

22.9 Summary

Table 22.7: *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 22.7 summarizes the commands used within this section.

Table 22.8: *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.

Table 22.8 summarizes the keywords and commands used within this section.

22.10 Practices

22.11 Practices

Exercise 22.1– 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 22.2– 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 22.3– Develop a script that calculates the square root of two cathetus. Use a function with local variables and arithmetic expansions.

Exercise 22.4– 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 22.5– 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 22.6– Using a function recursively, develop a script to calculate the factorial of a number.

Chapter 23

System Administration

Chapter 24

System Administration

24.1 Users Accounts

An account provides the user with configuration settings and preferences and also with some space in disk (typically 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 for users and groups of users which usually have 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. Examples of such users are www-data, lp, etc.

  Note. Some services run using the `root` user but the use of special users is preferred for security reasons.

The minimal data to create a user account is the name of the user or User ID, a password and a personal directory or home.

24.2 Configuration files

The configuration information of the users is saved in the following files:

```
/etc/passwd
/etc/shadow
/etc/group
/etc/gshadow
/etc/skel
```

Example text lines of the file `/etc/passwd` could be the following:

```
user1:x:1000:1000:Mike␣Smith,,,:/home/user1:/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. `user1`: 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.

6. `/home/user1`: 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:

```
user1: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

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:user1,user2
```

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. Groups can also have an associated password (although this is not very common). Typically, the encrypted password of the group is stored in another file called `/etc/gshadow`.

Another interesting configuration directory is `/etc/skel`. This directory contains the "skeleton" that is copied to each user's home when the user is created.

24.3 User Management

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.
- `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 24.4 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.

Finally, the command `chown` can be used to change the owner and the group of a file. Example:

```
# chown user1 notes.txt
```

The above command sets a new owner (user1) 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:

```
# chgrp -R students directory1
```

The command above sets the group "student" to "directory1" and all its contents.

24.4 Su and Sudoers

The commands `sudo` and `su` allow access to other commands as a different user.

The `sudo` command stands for "superuser do". It prompts you for your personal password and confirms your request to execute a command by checking a file, called sudoers (in /etc/sudoers), which the system administrator configures. Using the sudoers file, system administrators can give certain users or groups access to some or all commands without those users having to know the root password. 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 in the sudoers for any command you can get a "root shell" using:

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

On the other hand, the `su` command stands for "switch user", and allows you to become another user. To use the `su` command on a per-command basis, enter:

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

Replace user with the name of the account which you'd like to run the command as, and command with the command you need to run as another user. To switch users before running many commands, enter:

```
$ su user
```

Replace user with the name of the account which you'd like to run the commands as. The user feature is optional; 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, you'll be prompted for the password associated with the account for which you're trying to run the command. If you supply a user, you will be logged in as that account until you exit it. To do so, press Ctrl-d or type exit at the command prompt.

Using `su` creates security hazards, is potentially dangerous, and requires more administrative maintenance. It's not good practice to have numerous people knowing and using the root password because when logged in as root, you can do anything to the system. Notice that, on the other hand, the `sudo` command makes it easier to practice the principle of least privilege.

24.5 Installing Software

To fully understand the process of installing software in our Linux box, we have to understand static and dynamic libraries. Static libraries or statically-linked libraries are a set of routines, external functions and variables which are resolved in a caller at compile-time and copied into a target application by a compiler, linker, or binder, producing an object file and a stand-alone executable[1]. This executable and the process of compiling it are both known as a static build of the program.

On the other hand, with dynamic library linking, executable libraries of machine instructions are shared across packages and applications. In these systems, complex relationships between different packages requiring different versions of libraries results in a challenge colloquially known as "dependency hell". On Microsoft Windows systems, this is also called "DLL hell" when working with dynamically linked libraries. Good **Package Management Systems** become vital on these systems.

The main advantage of static libraries is that the application can be certain that all its libraries are present and that they are the correct version. This avoids dependency problems. 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. On Microsoft Windows it is common to include the library files an application needs with the application. On Unix-like systems this is less common as package management systems can be used to ensure the correct library files are available. This allows the library files to be shared between many applications leading to space savings. It also allows the library to be updated to fix bugs and security flaws without updating the applications that use the library.

Generally for Linux packages, the executables are copied to /usr/bin, the libraries to /usr/lib and the docs to /usr/share/doc/$package/. You may think that this is impossible to manage, with files sprinkled across the filesystem like this. However the **Package Management System** tracks where all the files are, allowing the user to easily uninstall etc. Also one can easily see the dependencies between packages, and are automatically told if an operation on a package would conflict with the rest of the system.

It's worth noting here that to take full advantage of the package management system, one should not go installing or deleting files behind its back. For e.g. only install from source as a last resort. In the unlikely event you can't find a package on the net, it's not too hard to make the package yourself.

There are multiple different package management systems in the linux world, the two main ones being Red Hat Packet Management (RPM) and Debian Packages (DPKG).

On the other hand, the **Package Management System** on linux has two levels:

- **Individual packages management.** Individual packages are managed by `rpm` and `dpkg` on Red Hat and Debian respectively. These two are functionally equivalent though, and will be compared in Table 24.1.

  Note. In the case of debian, APT uses a file that lists the 'sources' from which packages can be obtained. This file is /etc/apt/sources.list.

- **Package Management Systems.** At a higher level, package dependencies can be automatically managed by `yum` and `apt` on Red Hat and Debian respectively. These two are functionally equivalent though, and will be compared in Table 24.2. With these tools one can essentially say "install this package" and all dependent

[1] Historically, libraries could only be static.

Table 24.1: `dpkg` and `rpm`.

Debian	Red Hat	Description
`dpkg -Gi package(s).deb`	`rpm -Uvh packages(s).rpm`	install/upgrade package file(s)
`dpkg -r package`	`rpm -e package`	remove package
`dpkg -l '*spell*'`	`rpm -qa '*spell*'`	show all packages whose names contain the word spell
`dpkg -l package`	`rpm -q package`	show version of package installed
`dpkg -s package`	`rpm -q -i package`	show all package metadata
`dpkg -I package.deb`	`rpm -q -i -p package.rpm`	show all package file's metadata
`dpkg -S /path/file`	`rpm -q -f /path/file`	what package does file belong
`dpkg -L package`	`rpm -q -l package`	list where files were installed
`dpkg -c package.deb`	`rpm -q -l -p package.rpm`	list where files would be installed
`dpkg -x package.deb`	`rpm2cpio package.rpm | cpio -id`	extract package files to current directory
`dpkg -s package | grep ^Depends:`	`rpm -q --requires package`	list files/packages that package needs
`dpkg --purge --dry-run package`	`rpm -q --whatrequires package`	list packages that need package (see also whatrequires)

packages will be installed/upgraded as appropriate. One of course has to configure where these tools can find these packages, and this is typically done by configuring **online package repositories**.

Table 24.2: `apt` and `yum`.

Debian	Red Hat	Description
`apt-get dist-upgrade`	`yum update [package list]`	upgrade specified packages (or all installed packages if none specified)
`apt-get install <package list>`	`yum install <package list>`	install latest version of package(s)
`apt-get remove <package list>`	`yum remove <package list>`	remove specified packages from system
`apt-cache list [package list]`	`yum list [package list]`	list available packages from repositories

Note. Debian requires one to `apt-get update` before these commands so the local cache is up to date with the online ones. `Yum` is the opposite, in that one needs to add the -C option to tell it to operate on the local cache only.

Another very useful tool, which is not installed by default is apt-file. This tool allows us to search between packages and files. For example:

```
$ apt-file search /etc/init/ssh.conf
openssh-server: /etc/init/ssh.conf
```

Final remark. OS which do not use a package management system (like windows) traditionally release programs with an executable installer that contains any required dependencies (libraries etc.). There are many problems with this method, the most obvious being:

1. No standard install API so can't do many things like seeing where a file came from etc.
2. Size of install files large due to included dependencies and install logic.
3. Large burden on developers to develop install logic.

In the particular case of Microsoft, they have addressed these problems with MSI files, which are essentially the same as linux packages. However MSI files are still autonomous, i.e. they don't handle dependencies and so still suffer from the second problem above. The windows filesystem layout generally puts all files associated with a program under one directory (usually in "Program files"). There are many problems with this also, the main one being that this leads to mixing of logic and data which rather inflexible when for instance, we want to use different disk partitions for data and logic.

24.6 System Logging

24.6.1 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.

24.6.2 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 24.3: 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/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. You can rotate log file using the command `logrotate`.

24.6.3 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).

24.6.4 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, and it is possible to

have a single host that accepts all of the 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.

24.6.5 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.

24.6.6 Actions

The action field of a rule describes the abstract term "logfile". A "logfile" need not to be a real file but also a named pipe, a virtual console or a remote machine. To forward messages to another host, prepend the hostname with the at sign "@".

24.6.7 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
```

This 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.
- 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.

24.6.8 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.

24.6.9 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:

```
user1$ logger System rebooted
user1$ tail -1 /var/log/syslog
Sep  7 11:28:02 XPS user1: 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
```

24.7 Extra

24.7.1 *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.

24.7.2 *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.

24.7.3 *Example of a sudoers file

```
# 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
```

24.7.4 *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
```

24.8 Command summary

The table 24.4 summarizes the commands used within this section.

Table 24.4: *Commands.*

`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.

24.9 Practices

24.10 Practices

Exercise 24.1– 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

More Network Applications

Chapter 25

X window system

Chapter 26

X window system

26.1 What is the X service?

The X service[1], also known as X or X11 (11 is the current major version) is a software system and a network protocol to provide a basis for graphical user interfaces (GUIs) and rich input device capability. The X window system creates a hardware abstraction layer where software is written to use a generalized set of commands, allowing for device independence and reuse of programs on any computer that implements X. X uses a client/server model, in which the X server is built as a user space application and it provides an additional abstraction layer on top of the Kernel for its clients.

X does not mandate any particular user interface. Individual X client programs known as window managers handle this. There are a lot of window managers, the most widely known being GNOME and KDE. However, the window manager is not a requirement and programs may use X's graphical abilities without any user interface. The X server is often used in conjunction with an X session manager to implement sessions. In the X Window System, an X session manager is a program that can save and restore the current state of a set of running applications. A session is a "state of the desktop" at a given time: a set of windows with their current content. More precisely, a session is the set of clients managing these windows or related to them and the information that allows these applications to restore the condition of these windows if required. The most recognizable effect of using a session manager is the possibility of logging out from an interactive session and then finding exactly the same windows in the same state when logging in again.

Unlike most earlier display protocols, X was specifically designed to be used over network connections rather than on an integral or attached display device. X features the so-called "network transparency", which means that the machine where an application program (X client) runs can differ from the user's local machine (X server). The X server can communicate with various X clients. The server accepts requests for graphical output (windows) and sends back user input (from keyboard, mouse, or touchscreen). The user's computer running the server and the remote applications being the clients, often confuses novice people, because the terms appear reversed. But X takes the perspective of the application, rather than that of the end-user: X provides display and I/O services to applications, so it is a server; applications use these services, thus they are clients (see Figure 26.1).

26.2 Setting the display: using Unix sockets or TCP/IP sockets

The client/server model of X can use a number of 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.

[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.

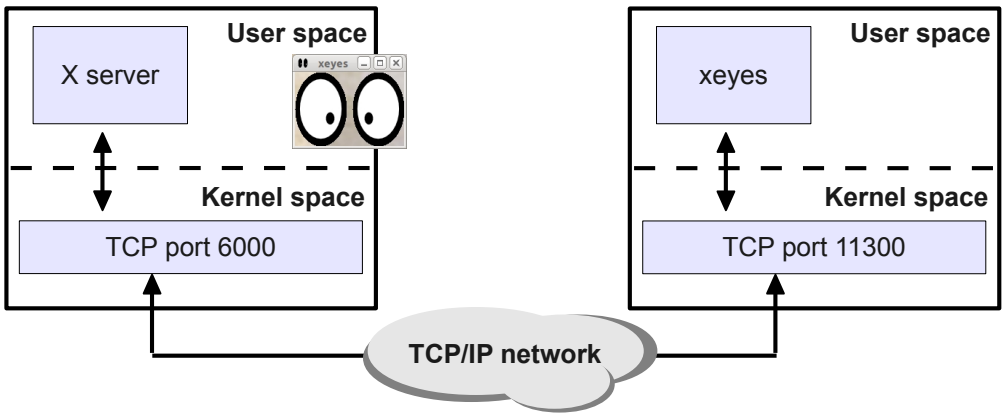


Figure 26.1: X client/server example.

- `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 (even localhost as in the example) via TCP/IP sockets. Another 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
```

26.3 X authentication mechanism

Finally, when an X server gets a connection on its port, there is a small amount of security that the client has to be checked on. 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 26.1).

For example, to enable access to our X server from host 192.168.0.1, we should type:

Table 26.1: *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 acccess control (all remote hosts will have access to X server).
`xhost -`	Turns access control back on.

```
$ xhost +192.168.0.1
```

We must notice that for security reasons, options that affect access control may only be run from the controlling host, i.e. the machine with the display connection.

26.4 Activate TCP sockets for the X server

In many modern Linux distributions the default X server does not listen to TCP/IP connections by default so you have to enable this feature.

In new Linux distros (like Ubuntu >= 11.10) the display manager is LightDM. To enable TCP connections to the X server in this case, open /etc/lightdm/lightdm.conf and add the following line to section [SeatDefaults]:

```
xserver-allow-tcp=true
```

Then, reboot the system.

Chapter 27

Secure Shell

Chapter 28

Secure Shell

28.1 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 28.5). The `sshd` daemon listens to port 22 by default. SSH encrypts all traffic (including passwords), which effectively eliminates eavesdropping, connection hijacking, and other attacks. The main use of SSH is for connecting to remote shells like TELNET.

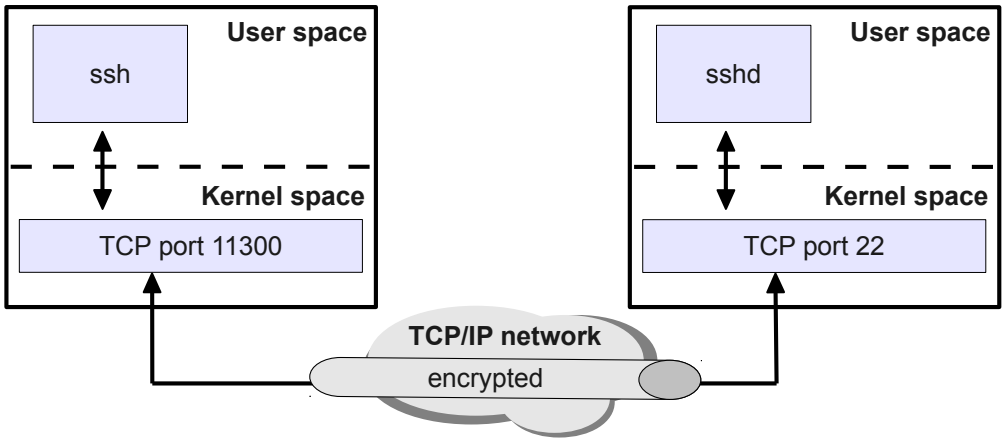


Figure 28.1: SSH client/server example.

28.2 SSH Architecture

SSH is a protocol that allows secure remote login and other secure network services over an insecure network. The main components of SSH architecture are three:

- the Transport Layer Protocol
- the User Authentication Protocol
- the Connection Protocol

The first component provides server authentication, confidentiality, integrity and optionally compression. The transport layer will typically be run over the top of any reliable data stream, in particular over TCP/IP connection.

The second one, running over SSH transport layer protocol, authenticates the client-side user to the server. Finally, the third one, running over user authentication protocol, multiplexes the encrypted tunnel into several logical channels (see figure 28.2).

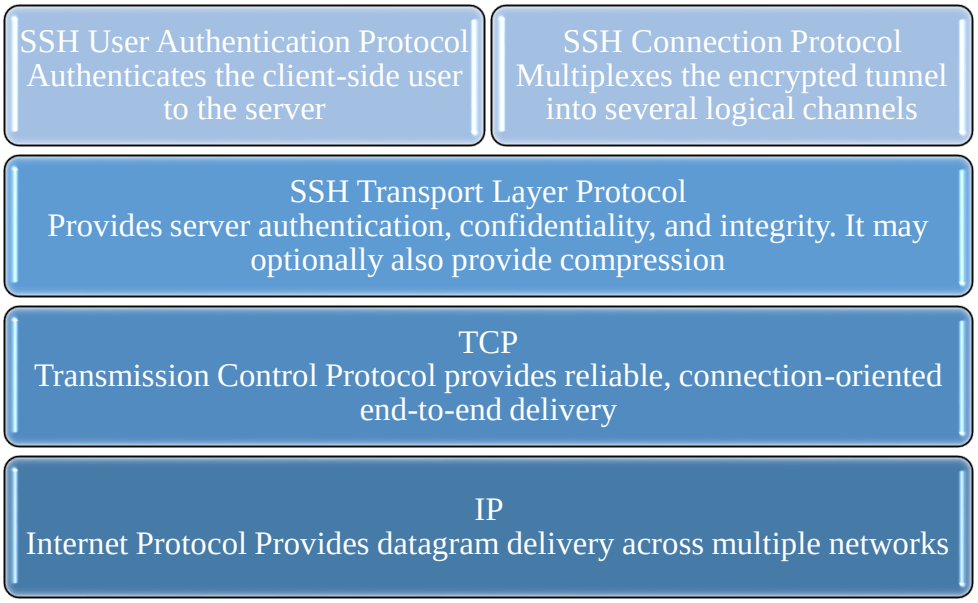


Figure 28.2: SSH protocol stack.

The SSH communication works on top of the packet-level protocol, and proceeds in the following phases:

1. The client opens a connection to the server.
2. The server sends its public host key and another public key ("server key") that changes every hour. The client compares the received host key against its known host keys.
3. The client generates a 256 bit random number using a cryptographically strong random number generator, and chooses an encryption algorithm from those supported by the server. The client encrypts the random number (session key) with a public key algorithm, using both the host key and the server key, and sends the encrypted key to the server.
4. The server decrypts the public key encryptions and recovers the session key. Both parties start using the session key (until this point, all traffic has been unencrypted on the packet level). The server sends an encrypted confirmation to the client. Receipt of the confirmation tells the client that the server was able to decrypt the key, and thus holds the proper private keys. At this point, the server machine has been authenticated, and transport level encryption and integrity protection are in use.
5. The user is authenticated to the server. The client sends requests to the server. The first request always declares the user name to log in as. The server responds to each request with either "success" (no further authentication is needed) or "failure" (further authentication is required).
6. After the user authentication phase, the client sends requests that prepare for the actual session. Such requests include allocation of a tty, X11 forwarding, TCP/IP forwarding, etc. After all other requests, the client sends a request to start the shell or to execute a command. This message causes both sides to enter the interactive session.
7. During the interactive session, both sides are allowed to send packets asynchronously.

The packets may contain data, open requests for X11 connections, forwarded TCP/IP ports, or the agent, etc. Finally at some point the client usually sends an EOF message. When the user's shell or command exits, the server sends its exit status to the client, and the client acknowledges the message and closes the connection

28.2.1 Transport Layer Protocol

Server authentication occurs at the transport layer, based on the server possessing a public-private key pair. A server may have multiple host keys using multiple different asymmetric encryption algorithms. Multiple hosts may share the same host key. In any case, the server host key is used during key exchange to authenticate the identity of the host. For this authentication to be possible, the client must have presumptive knowledge of the server public host key. RFC 4251 dictates two alternative trust models that can be used:

- The client has a local database that associates each host name (as typed by the user) with the corresponding public host key. This method requires no centrally administered infrastructure and no third-party coordination. The downside is that the database of name-to-key associations may become burdensome to maintain.

- The host name-to-key association is certified by a trusted Certification Authority (CA). The client knows only the CA root key and can verify the validity of all host keys certified by accepted CAs. This alternative eases the maintenance problem, because ideally only a single CA key needs to be securely stored on the client. On the other hand, each host key must be appropriately certified by a central authority before authorization is possible.

The SSH transport layer is a secure low level transport protocol, that works normally over TCP/IP listening for connections on port 22. It supplies strong encryption, cryptographic host authentication, and integrity protection. Authentication is host-based, in fact it's the higher level protocol to provide user authentication. The SSH transport layer permits negotiation of key exchange method, public key algorithm, symmetric encryption algorithm, message authentication algorithm, and hash algorithm. Furthermore it minimizes the number of round-trips that will be needed for full key exchange, server authentication, service request, and acceptance notification of service request. The number of round-trips is normally two, but it becomes three in the worst case. The subphases of this protocol are three and the sequence is: compression, MAC and encryption. During the exchanging period each packet has the format of table 28.1 (see Figure 28.3). The payload contains the exchanging data, which may be compressed. The random padding is necessary to make become the length of the string $(packet\_length||padding\_length||payload||padding)$ divisible by the cipher block size or 8.

Table 28.1: Packet format in the exchanging phase

uint32	packet_length
byte	padding_length
byte[n_1]	payload
byte[n_2]	Random padding
byte[m]	mac

where $n_1 = packet\_length - padding\_length - 1$, $n_2 = padding\_length$, $m = mac\_length$.

Figure 28.4 illustrates the sequence of events in the SSH Transport Layer Protocol. First, the client establishes a TCP connection to the server with the TCP protocol and is not part of the Transport Layer Protocol. When the connection is established, the client and server exchange data, referred to as packets, in the data field of a TCP segment.

Compression algorithm may be different for each direction of communication. To guarantee data integrity, each packet includes a MAC. The MAC phase is not included at the first time when its length is zero. Only after the key exchange, MAC will be in effect. The MAC is calculated from a shared secret, packet sequence number and part of packet (the length fields, payload and padding):

$$MAC = mac(key, sequence\_number||unencrypted\_packet)$$

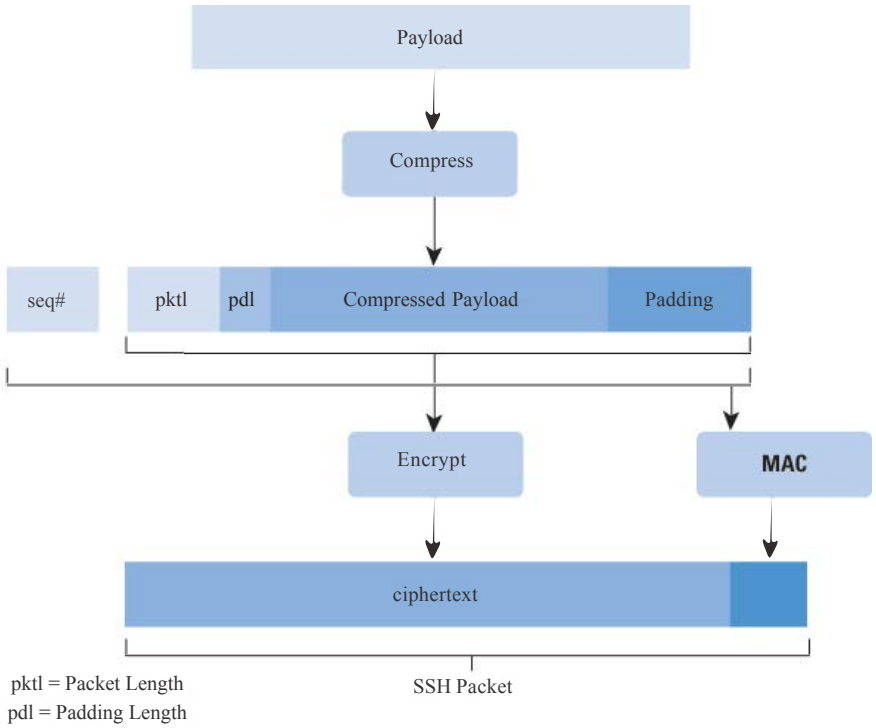


Figure 28.3: SSH Transport Layer Protocol Packet Formation.

Table 28.2: SSH classification of MAC algorithms.

Name		**Explanation**
hmac-sha1	REQUIRED	HMAC-SHA1 (digest length = key length =20)
hmac-sha1-96	RECOMMENDED	first 96 bits of HMAC-SHA1 (digest length = 12, key length = 20)
hmac-md5	OPTIONAL	HMAC-MD5 (digest length = key length = 16)
hmac-md5-96	OPTIONAL	first 96 bits of HMAC-MD5 (digest length = 12, key length = 16)
none	OPTIONAL	no MAC; NOT RECOMMENDED

number range begins from zero and ends to (2^{32}-1); the packet sequence number is not sent inside the exchanged packets. The MAC algorithm (see table 28.2) may be different for each direction of communication.

Finally, the string

$$(packet\_length||padding\_length||payload||padding)$$

is encrypted. It is worth noting that also the packet length field is encrypted. This trick is adopted to hide the dimension of payload, because a part of exchanged packet is random padding, the other one is the exchanged datas. The minimum size of a packet is 16 or the cipher block size bytes plus MAC length. The maximum size is respectively of 32768 bytes for uncompressed payload length and of 35000 bytes for total packet. It's specified the dimension of uncompressed payload, because this field, and only it, may be compressed using an appropriate algorithm. In the encryption phase the packet length, padding length, payload and padding fields of each packet are encrypted with negotiated algorithm and key. Encryption algorithm (see table 28.3) may be different for each direction of communication, in particular all encrypted packets sent in one direction is considered as single data stream. It is recommended to not use key length less than 128 bits.

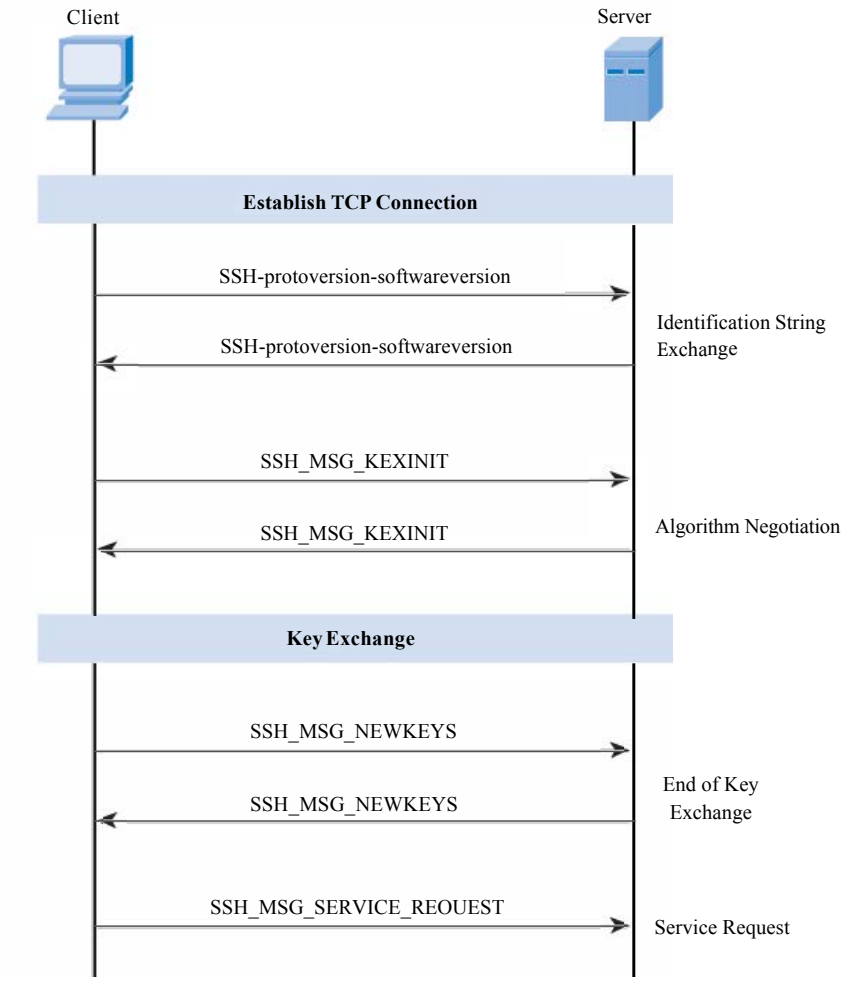


Figure 28.4: SSH Transport Layer Protocol.

Key Exchange Methods

In the following we describe the key exchange method at the beginning of that period. During the first connection, the key exchange method indicates the generation of keys for encryption and authentication, and how the server authentication is done. The only required key exchange method is Diffie-Hellman. This protocol has been designed to be able to operate with almost any public key format, encoding, and algorithm (signature and/or encryption); see table 28.4.

At the beginning of key exchange each side sends a list of supported algorithms (see Table 28.3). Each side has an own algorithm in each category and tries to guess that one of the other side. The guess may be wrong if kex algorithm and/or the host key algorithm are different for server and client or if any of the listed algorithms cannot be agreed upon. Otherwise, the guess is right and the sent packet are handled as the first key exchange packet. Before beginning the communication, the client waits for response to its service request message.

Encryption and authentication keys are derived, after the key exchange by an exchange hash H and a shared secret K. From the first key exchange, the hash H is used additionally as the session identifier, in particular as authentication method. In fact this is the proof of possession of the right private key. The exchange hash H lives for the session duration, and it will not be changed. Each key exchange method specifies a hash function that is used in the key exchange to compute encryption keys:

Table 28.3: SSH classification of cipher algorithms.

Name		Explanation
3des-cbc	REQUIRED	three-key 3DES in CBC mode
blowfish-cbc	RECOMMENDED	Blowfish in CBC mode
twofish256-cbc	OPTIONAL	Twofish in CBC mode, with 256-bit key
twofish-cbc	OPTIONAL	alias for "twofish256-cbc"
twofish192-cbc	OPTIONAL	Twofish with 192-bit key
twofish128-cbc	RECOMMENDED	Twofish with 128-bit key
aes256-cbc	OPTIONAL	AES (Rijndael) in CBC mode, with 256-bit key
aes192-cbc	OPTIONAL	AES with 192-bit key
aes128-cbc	RECOMMENDED	AES with 128-bit key
serpent256-cbc	OPTIONAL	Serpent in CBC mode, with 256-bit key
serpent192-cbc	OPTIONAL	Serpent with 192-bit key
serpent128-cbc	OPTIONAL	Serpent with 128-bit key
arcfour	OPTIONAL	the ARCFOUR stream cipher
idea-cbc	OPTIONAL	IDEA in CBC mode
cast128-cbc	OPTIONAL	CAST-128 in CBC mode
none	OPTIONAL	no encryption; NOT RECOMMENDED

Table 28.4: public key and/or certificate formats.

Name		Explanation
ssh-dss	REQUIRED	Simple DSS
ssh-rsa	RECOMMENDED	Simple RSA
x509v3-sign-rsa	OPTIONAL	X.509 certificates(RSA key)
x509v3-sign-rsa	OPTIONAL	X.509 certificates(DSS key)
spki-sign-rsa	OPTIONAL	SPKI certificates(RSA key)
spki-sign-dss	OPTIONAL	SPKI certificates (DSS key)
pgp-sign-rsa	OPTIONAL	OpenPGP certificates(RSA key)
pgp-sign-dss	OPTIONAL	OpenPGP certificates(DSS key)

Table 28.5: Key exchange packet.

byte	`SSH_MSG_KEXINIT`
byte[16]	cookie (random bytes)
string	kex_algorithms
string	server_host_key_algorithms
string	encryption_algorithms_client_to_server
string	encryption_algorithms_server_to_client
string	mac_algorithms_client_to_server
string	mac_algorithms_server_to_client
string	compression_algorithms_client_to_server
string	compression_algorithms_server_to_client
string	languages_client_to_server
string	languages_server_to_client
boolean	first_kex_packet_follows
uint32	0 (reserved for future extension)

- Initial IV client to server: HASH(K || H || “A” || session_id)
- Initial IV server to client: HASH(K || H || “B” || session_id)
- Encryption key client to server: HASH(K || H || “C” || session_id)
- Encryption key server to client: HASH(K || H || “D” || session_id)
- Integrity key client to server: HASH(K || H || “E” || session_id)
- Integrity key server to client: HASH(K || H || “F” || session_id)

Where K is a known value and “A” as byte and `session_id` as raw data. Key data is taken from the beginning of the hash output, at least 128 bits (16 bytes). Key exchange ends by each side sending an `SSH_MSG_NEWKEYS` message. This message is sent with the old keys and algorithms. All messages sent after this message must use the new keys and algorithms.

The shared secret K is provided by the Diffie-Hellman key exchange. Each side is not able to determine alone a shared secret. To supply host authentication, the key exchange is combined with a signature with the host key. In the following description: p is a large safe prime, g is a generator for a subgroup of GF(p), and q is the order of the subgroup; V_S is server’s version string; V_C is client’s version string; K_S is server’s public host key; I_C is client’s KEXINIT message and I_S server’s KEXINIT message which have been exchanged before this part begins.

- Client generates a random number x ($1 < x < q$) and computes $e = g^x \bmod p$. C sends “e” to server.
- Server generates a random number y ($0 < y < q$) and computes $f = g^y \bmod p$. Server receives “e”. It computes $K = e^y \bmod p$, H = hash(V_C || V_S || I_C || I_S || K_S || e || f || K) (table 4), and signature s on H with its private host key. Server sends “K_S || f || s” to C. The signing operation may involve a second hashing operation.
- Client verifies that K_S really is the host key for Server (e.g. using certificates or a local database). Client is also allowed to accept the key without verification (insecure). Client then computes $K = f^x \bmod p$, H = hash(V_C || V_S || I_C || I_S ||K_S || e || f || K), and verifies the signature s on H.

Either side must not send or accept “e” or “f” values that are not in the range [1, p-1]. If this condition is violated, the key exchange fails.

Table 28.6: packet from client to server.

byte	`SSH_MSG_KEXDH_INIT`
mpint	e

Table 28.7: packet from server to client.

byte	`SSH_MSG_KEXDH_REPLY`
string	server public host key and certificates (K_S)
mpint	f
string	signature of H

It’s possible re-exchange keys and above all it’s recommended when the communication transmitted more than a gigabyte of data or it is during more than an hour. However, the re-exchange keys operation requires computational resources, so it should not be performed too often. This operation is equal to the initial key exchange: it’s possible to negotiate some or all of the algorithms, all keys and initialization vectors are recomputed after the exchange and compression and encryption contexts are reset. The only thing that remains unchanged is the session identifier.

After key exchange, the client requests a service, passing through SSH user authentication protocol. The service is identified by a name. When service end, user may want to terminate the connection. After this message the client and server must not send or receive any data.

Table 28.8: hash components.

Type	Name	Explanation
string	V_C	the client's version string (CR and NL excluded)
string	V_S	the server's version string (CR and NL excluded)
string	I_C	the payload of the client's `SSH_MSG_KEXINIT`
string	I_S	the payload of the server's `SSH_MSG_KEXINIT`
string	K_S	the host key
mpint	e	exchange value sent by the client
mpint	f	exchange value sent by the server
mpint	K	the shared secret

28.2.2 User Authentication Protocol

Authentication, also referred to as user identity, is the means by which a system verifies that access is only given to intended users and denied to anyone else. Many authentication methods are currently used, ranging from familiar typed passwords to more robust security mechanisms. Most Secure Shell implementations include password and public key authentication methods but others (e.g. kerberos, NTLM, and keyboard interactive) are also available. The Secure Shell protocol's flexibility allows new authentication methods to be incorporated into the system as they become available. Once the identity of the remote SSH server has been verified, the SSH client sends the user name of the user who is requesting a login, along with any credentials (based on type of authentication) of the user to the target SSH server. The user is authenticated in one of three ways:

- Public key authentication — using key files
- Using host-based authentication — using key files
- Password authentication

Each method is attempted by the SSH client in sequence until there is a successful user authentication or after the last method is tried with no response and results in a failure. Public key authentication is the most secure way to authenticate a user. Each user has a private key to identify that user, which is kept secret at the client. A corresponding public key is used by anyone wishing to authenticate this user, and the SSH server has a copy of this public key. The SSH client uses the user's private key to encrypt a message sent to the SSH server. The SSH server uses the user's public key to decrypt this message. If this is successful, the user is authenticated because the message must have been sent by a client with access to the private Host Based authentication is very similar to the user public key authentication, and is also based on public and private keys. In this case, separate keys are not used for each user. Instead, a single key pair is used to authenticate the SSH client system and the SSH server trusts the client as to the identity of the individual users. The SSH client uses the client system's private key to encrypt a message to the server, and the SSH server uses the public key for that client system (host) to decrypt the message. If this is successful, the user supplied by the client is authenticated. Password authentication uses the familiar mechanism to authenticate a user. The user name and password are sent over the encrypted channel to the SSH server, which authenticates the us er using the supplied password.

The SSH authentication protocol is a general-purpose user authentication protocol, running over the SSH transport layer protocol. At the beginning this protocol receives the session identifier from the lower-level protocol to identify session and to prove ownership of a private key. The SSH authentication protocol expects that SSH transport layer protocol provides integrity and confidentiality protection. The server begins the authentication asking to the client which authentication methods can be used to continue the exchange at any given time. The client can choose the methods listed by the server in any order. The server has a timeout for authentication and a maximum number of failed authentication attempts. An user authentication protocol session provides for an authentication request and response.

Authentication Requests

In table 28.9 it is shown the authentication request message format. The user and service name appear in every new authentication attempt. These may change; therefore the server checks them in every message. If there are some changes the server must update authentication states. If it is unable to make it, it disconnects the user for that service.

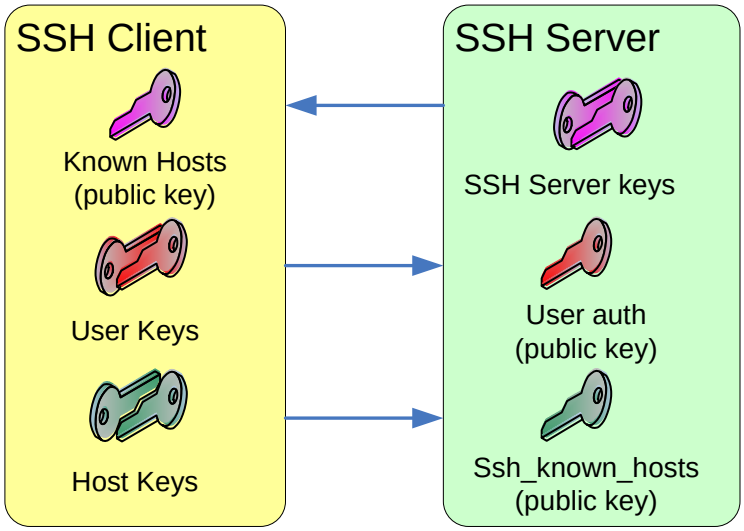


Figure 28.5: User Authentication methods.

In particular the user name specifies the user identity and the service name that he wants to start after authentication. If the user name does not exist, the server may send a list of acceptable authentication methods, but in any case the authentication request is rejected avoiding to disclose information on existing logins. If the requested service is not available or does not exist, the server may disconnect the user for that service, sending it an apposite disconnect message. The last defined field is the name of authentication method; the remaining ones depend on the authentication method. In particular an authentication request may result in a further exchange of messages.

Table 28.9: authentication request message format.

byte	`SSH_MSG_USERAUTH_REQUEST`
string	user name
string	service name
string	method name
The rest of the packet is method-specific.	

Authentication Responses

In table 28.10 it is depicted the rejection of authentication request message. In “Authentications that can continue” there is a list of authentication method names with which it’s possible to continue the authentication dialog.

Table 28.10: rejection of authentication request message.

byte	`SSH_MSG_USERAUTH_FAILURE`
string	authentications that can continue
boolean	partial success

When the server accepts authentication and only when the authentication is complete, it responds with the message of table 28.11.

The client is not obligated to wait for responses from previous requests, so it may send several authentication requests. Otherwise the server must process each request before processing the next request. It is not possible to send a second request without waiting for a response from the server. In fact if the first request will result in further

Table 28.11: acceptance of authentication request message.

byte	SSH_MSG_USERAUTH_SUCCESS

exchange of messages, that one will be aborted by a second request. No SSH_MSG_USERAUTH_FAILURE message will be sent for the aborted method. SSH_MSG_USERAUTH_SUCCESS is sent only once any further authentication requests received after that is silently ignored, passing directly that request to the service being run on top of this protocol. It's possible for a client to use the "none" authentication method. If no authentication is needed for user, server returns SSH_MSG_USERAUTH_SUCCESS. Otherwise, server returns SSH_MSG_USERAUTH_FAILURE.

Authentication Method

In the authentication at the lower-level it's been created a secure channel between the client and the server, but different user may run this operation from the same client to the same server; so it's required, as user authentication method. The main user authentication methods are two: a public key or a password authentication. In the first method the possession of a private key serves as authentication. The user sends a signature created with its private key. The server checks that the public key and the signature for the user are valid. If it's valid, the authentication request is accepted; otherwise it is rejected. In table 28.12 it is shown the message format for querying whether authentication using the key would be acceptable.

Table 28.12: message format for querying whether authentication using the key would be acceptable.

byte	SSH_MSG_USERAUTH_REQUEST
string	user name
string	Service
string	"publickey"
boolean	FALSE
string	public key algorithm name
string	public key blob

Public key algorithms are defined in the transport layer specification (table 28.4); in particular the list is not constrained by what was negotiated during key exchange. The public key blob may contain certificates. The server responds to this message with either SSH_MSG_USERAUTH_FAILURE or with the SSH_MSG_USERAUTH_PK_OK (table28.13). Public key authentication is one of the most secure methods to authenticate using Secure Shell. Public key authentication uses a pair of computer generated keys – one public and one private. Each key is usually between 1024 and 2048 bits in length, and appears like the sample below. Even though you can see it, it is useless unless you have the corresponding private key:

```
---- BEGIN SSH2 PUBLIC KEY ----
Subject:
Comment: my public key
AAAAB3NzaC1kc3MAAACBAKoxPsYlv8Nu+fncH2ouLiqkuUNGIJo8iZaHdpDABAvCvLZn
jFPUN+SGPtzP9XtW++2q8khlapMUVJS0OyFWgl0ROZwZDApr2olQK+vNsUC6ZwuUDRPV
fYaqFCHrjzNBHqgmZV9qBtngYD19fGcpaq1xvHgKJFtPeQOPaG3Gt64FAAAAFQCJfkGZ
e3alvQDU8L1AVebTUFi8OwAAAIBk9ZqNG1XQizw4ValQXREczlIN946Te/1pKUZpau3W
iiDAxTFlK8FdE2714pSV3NVkWC4xlQ3x7wa6AUXIhPdLKtiUhTxtctm1epPQS+RZKrRI
XjwKL71EO7UY+b8EOAC2jBNIRtYRy0Kxsp/NQ0YYzJPfn7bqhZvWC7uiC+D+ZwAAAIEA
mx0ZYo5jENA0IinXGpc6pYH18ywZ8CCI2QtPeSGP4OxxOusNdPskqBTe5wHjsZSiQr1g
b7TCmH8Tr50Zx+EJ/XGBU4XoWBJDifP/6Bwryejo3wwjh9d4gchaoZNvIXuHTCYLNPFo
RKPx3cBXHJZ27khllsjzta53BxLppfk6TtQ=
---- END SSH2 PUBLIC KEY ---
```

Table 28.13: acceptance of authentication message.

byte	SSH_MSG_USERAUTH_PK_OK
string	public key algorithm name from the request
string	public key blob from the request

To perform actual authentication, the client may send the signature, generated using its private key, directly without first verifying whether the key is acceptable (table 28.14).

Table 28.14: message with signature.

byte	SSH_MSG_USERAUTH_REQUEST
string	user name
string	service
string	"publickey"
boolean	TRUE
string	public key algorithm name
string	public key to be used for authentication
string	signature

Signature is created by the corresponding private key over the data of table 28.15.

Table 28.15: data used in this order by signature.

string	session identifier
byte	SSH_MSG_USERAUTH_REQUEST
string	user name
string	service
string	"publickey"
boolean	TRUE
string	public key algorithm name
string	public key to be used for authentication

When the server receives this message, it checks whether the supplied key is acceptable for authentication, and if so, it checks whether the signature is correct. If both checks succeed, this method is successful. (Note that the server may require additional authentications.) The server responds with SSH_MSG_USERAUTH_SUCCESS (if no more authentications are needed), or SSH_MSG_USERAUTH_FAILURE (if the request failed, or more authentications are needed). In the second method, the possession of a password serves as authentication (see table 28.16). The server interprets the password and validates it against the password database; it may request user to change password. It's important to remember that if a cleartext password is transmitted in the packet, however this one is encrypted by SSH transport layer. Although passwords are convenient, requiring no additional configuration or setup for your users, they are inherently vulnerable in that they can be guessed, and anyone who can guess your password can get into your system. Due to these vulnerabilities, it is recommended that you combine or replace password authentication with another method like public key.

If the lower-level does not provide confidentiality, password authentication is disabled. Beside if the lower-level does not provide confidentiality or integrity password change is disabled. In general, the server responds to the password authentication packet with success or failure. In a failure response if the user password has expired server can reply to user with SSH_MSG_USERAUTH_PASSWD_CHANGEREQ (table 28.17). In any case server must prevent from logging users with expired password; then the client may continue with another authentication method, or request a new user password (table 28.18).

Finally the server must reply to request message with one of this message:

Table 28.16: Password authentication packet.

byte	SSH_MSG_USERAUTH_REQUEST
string	user name
string	service
string	"password"
boolean	FALSE
string	plaintext password

Table 28.17: Change request password.

byte	SSH_MSG_USERAUTH_PASSWD_CHANGEREQ
string	prompt
string	language tag

Table 28.18: retry password authentication .

byte	SSH_MSG_USERAUTH_REQUEST
string	user name
string	service
string	"password"
boolean	TRUE
string	plaintext old password
string	plaintext new password

- SSH_MSG_USERAUTH_SUCCESS: the password has been changed, and authentication has been successfully completed.
- SSH_MSG_USERAUTH_FAILURE with partial success: the password has been changed, but more authentications are needed.
- SSH_MSG_USERAUTH_FAILURE without partial success: the password has not been changed. Either password changing was not supported, or the old password was bad.
- SSH_MSG_USERAUTH_CHANGEREQ: the password was not changed because the new password was not acceptable.

28.2.3 Connection Protocol

The SSH Connection Protocol runs on top of the SSH Transport Layer Protocol and assumes that a secure authentication connection is in use. That secure authentication connection, referred to as a tunnel, is used by the Connection Protocol to multiplex a number of logical channels. All types of communication using SSH, such as a terminal session, are supported using separate channels. Either side may open a channel. For each channel, each side associates a unique channel number, which need not be the same on both ends. Channels are flow-controlled using a window mechanism. No data may be sent to a channel until a message is received to indicate that window space is available. The life of a channel progresses through three stages: opening a channel, data transfer, and closing a channel. The SSH stack we find the SSH Connection Protocol provides interactive login sessions, remote execution of commands, forwarded TCP/IP connections, and forwarded X11 connections. The requests may be several, different and may use independent channels. All such requests use the format of table 28.19 and the server answers success or failure with that one of tables 28.20 and 28.21.

In the channel mechanism all terminal sessions and forwarded connections are channels. Each side may open a channel. Multiple channels are multiplexed into a single connection. Channels are identified by numbers at each end (these numbers may be different on each side). Channels are flow-controlled; so no data may be sent to a channel until

Table 28.19: request format.

Byte	SSH_MSG_GLOBAL_REQUEST
String	request name
Boolean	want reply
... request-specific data follows	

Table 28.20: success response format failure response format.

Byte	SSH_MSG_REQUEST_SUCCESS
... request-specific data	

Table 28.21: failure response format.

Byte	SSH_MSG_REQUEST_FAILURE
... request-specific data	

a message is received to indicate that window space is available. To open a new channel, each side allocates a local number and sends the message of table28.22 to the other side including the local channel number and initial window size.

Table 28.22: message to open a channel

Byte	SSH_MSG_CHANNEL_OPEN
String	channel type
Uint32	sender channel
Uint32	initial window size
Uint32	maximum packet size
... channel type specific data follows	

In particular the fields of this message are defined following. Sender channel is a local sender identifier for the channel. Initial window size specifies how many bytes of channel data can be sent to the sender of this message without adjusting the window. Maximum packet size specifies the maximum size of an individual data packet that can be sent to the sender. On other side the remote machine decides whether it can open the channel, and it may respond with its open confirmation message (see table 28.23).

Table 28.23: message to confirm the opening of the channel

Byte	SSH_MSG_CHANNEL_OPEN_CONFIRMATION
Uint32	recipient channel
Uint32	sender channel
Uint32	initial window size
Uint32	maximum packet size
... channel type specific data follows	

In particular recipient channel is the channel number given in the original open request. Whether it can not open the channel, because does not support the specified channel type, it responds with SSH_MSG_CHANNEL_OPEN_FAILURE (see table 28.24). If the recipient of the SSH_MSG_CHANNEL_OPEN message does not support the specified channel type, it simply responds with SSH_MSG_CHANNEL_OPEN_FAILURE. After the opening of the channel some data is transmitted on this one. During the communication on channel, the data transfer is controlled by the window size that specifies how many bytes the other party can send before it must wait for the window to be adjusted.

The maximum amount of allowed data is the current window size. The amount of data sent decrements the window size. Each side may ignore all extra data sent after the allowed window is empty. When a side will no send more data

Table 28.24: message to fail the opening of the channel

Byte	SSH_MSG_CHANNEL_OPEN_FAILURE
Uint32	recipient channel
Uint32	reason code
String	additional textual information
String	language tag

to a channel, it send MSG_CHANNEL_EOF (see table 28.25), but no explicit response is sent to this message. It is important to note that the channel remains open after MSG_CHANNEL_EOF, and more data may be transferred in each direction.

Table 28.25: message to suspend a communication on a channel

Byte	SSH_MSG_CHANNEL_EOF
Uint32	recipient channel

Finally, when each side wants to terminate the channel, one sends a SSH_MSG_CHANNEL_CLOSE message (see table 28.26) and waits to receive back the same message. The channel is closed for each side when it has both sent and received SSH_MSG_CHANNEL_CLOSE.

Table 28.26: message to close the channel

Byte	SSH_MSG_CHANNEL_CLOSE
Uint32	recipient channel

Figure 28.6 provides an example of Connection Protocol Exchange.

Many channel types have extensions that are specific to that particular channel type (for example requesting a pseudo terminal for an interactive session). All channel-specific requests use the following format of table 28.28.

Table 28.27: channel-specific request format

Byte	SSH_MSG_CHANNEL_REQUEST
Uint32	recipient channel
String	Request type
Boolean	want reply
... type-specific data	

If want reply is FALSE, no response will be sent to the request. Otherwise, the recipient responds with either SSH_MSG_CHANNEL_SUCCESS or SSH_MSG_CHANNEL_FAILURE, or request-specific continuation messages. If the request is not recognized or is not supported for the channel, SSH_MSG_CHANNEL_FAILURE is returned. Request types are local to each channel type. To complete this section, it is defined a session as a remote execution of a program. The program may be an X11 Forwarding, a TCP/IP Port Forwarding, a Shell, a command, an application, a system command, or some built-in subsystem. Multiple sessions can be active simultaneously.

28.3 Practical SSH

While there are numerous SSH clients and servers, the most-used still remain the ones provided by OpenSSH. OpenSSH has been the default SSH client/server for every major Linux operation (including Ubuntu).

Let's start showing how SSH works with some examples. The first example is to use the ssh client to connect to a sshd server to obtain a remote terminal. You can achieve this by typing:

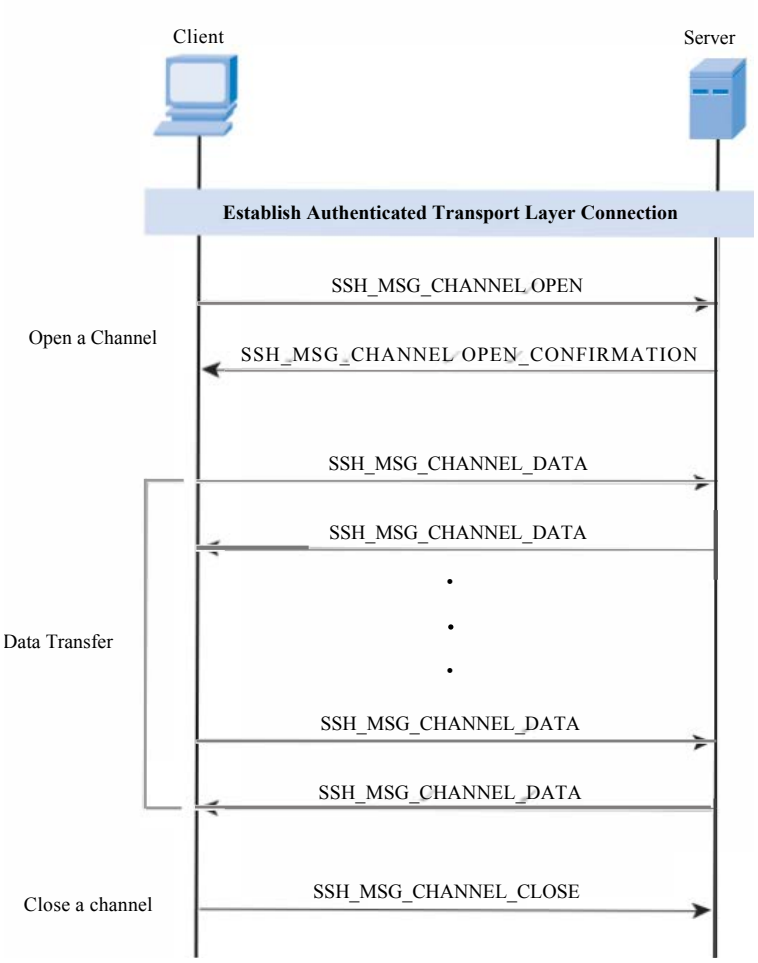


Figure 28.6: SSH Connection Protocol Message Exchange.

```
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 (Client) Configuration The default configuration file for `sshd` on most distributions is `/etc/ssh/ssh_config`. Key settings that one will want to check and possibly change:

- `ForwardAgent` – Specifies whether the connection to the authentication agent (if any) will be forwarded to the remote machine.
- `ForwardX11` – Specifies whether X11 connections will be automatically redirected over the secure channel. "yes" means that when one starts a GUI/X11 program from an ssh session, the program will run on the remote machine, but will display back on the machine the ssh client was started on. All traffic to the GUI program is encrypted, basically being carried in an ssh tunnel.
- `Protocol` – Determines which protocol clients can connect with. Should be only "2".

Table 28.28: OpenSSH Suite

`ssh`	The command-line client establishes encrypted connections to remote machines and will run commands on these machines if needed.
`scp`	To copy files (non-interactively) locally to or from remote computers, you can use `scp` ("secure copy").
`sftp`	The ftp client ("secure ftp") supports interactive copying. Like other command-line ftp clients, the tool offers many other options in addition to copying, such as changing and listing directories, modifying permissions, deleting directories and files, and more.
`sshd`	The SSH server is implemented as a daemon and listens on port 22 by default. SSH clients establish connections to the `sshd`.
`ssh-copy-id`	Nice little program for installing your personal identity key to a remote machine's `authorized_keys` file
`ssh-keygen`	Generates and manages RSA and DSA authentication keys.
`ssh-keysign`	The tool `ssh-keysign` is used for host-based authentication.
`ssh-keyscan`	This application displays public hostkeys and appends them to the `~/.ssh/known_hosts` file.
`ssh-agent`	The SSH agent manages private SSH keys and simplifies password handling. Remembers your passphrases over multiple SSH logins for automatic authentication. `ssh-agent` binds to a single login session, so logging out, opening another terminal, or rebooting means starting over.
`ssh-add`	The `ssh-add` tool introduces the `ssh-agent` to new keys.

sshd (Server) Configuration The default configuration file for `sshd` on most distributions is `/etc/ssh/sshd_config`. Key settings that one will want to check and possibly change:

- `MaxAuthTries` – Determines how many authentication failures are allowed before the connection closes. For more security, you can set this to 2 or even 1. Note, though, that while this can slow down password attacks, it cannot prevent them because it simply closes the current connection, and an attacker will then simply call ssh again to re-connect.

- `PasswordAuthentication` – Specifies whether password authentication is allowed. While the default is usually yes, security can be increased by setting this to "no" and using public key authentication.

- `PermitRootLogin` – Determines whether someone can attempt to login directly as root (administrator). For highest security, you should set this to: "no" . While this means that root will have to login as a normal user and then use su to become root, it can help prevent brute-force password attacks. This does make it impossible to run `scp` and `sftp` as root, which can be annoying—though it is certainly more secure. An alternative is to set it to "without-password" which will allow direct root login using public key authentication.

- `Protocol` – Determines which ssh protocol (1 and/or 2) clients are allowed to connect with. For security, you should make sure that this is just "2", not "1,2".

- `PubkeyAuthentication` – Specifies whether public key authentication is allowed.

- `X11Forwarding` – Specifies whether X11 forwarding is permitted. Usually yes in modern distributions.

sshd is usually configured to observe the access specifications in the standard server access control files `/etc/hosts.deny` and `/etc/hosts.allow`. These files provide an important mechanism to prevent automated brute force password attacks against an ssh server. The best security is provided by having `/etc/hosts.deny`contain the line: `ALL:ALL` and then having an `sshd` line in `/etc/hosts.allow`, e.g., like: `sshd: 192.168.1.1`; to permit logins from only certain ranges of IP addresses or hostnames.

The initial key discovery On first establishing contact, the other end of the connection reveals its public hostkey fingerprint. When warned that the authenticity of the machine has not been verified, you need to say yes, and then you will be prompted to enter the password. This initial key discovery process is to ensure security. It is possible for an attacker to steal information from the remote user log-in by impersonating the server, i.e., if the attacker can provide a server with the same host name and user authentication, the user connecting from the remote machine will be logged into a fraud machine and data may be stolen. Each server will have a randomly generated RSA server key. To ensure security, in cases where the server key changes, the SSH client will issue a serious warning reporting that the host identification has failed and that it will stop the log-in process.

The remote system's hostkeys are stored in the `~/.ssh/known_hosts` file. The next time you log in to the machine, SSH will check to see whether the key is unchanged and, if not, will refuse to cooperate: WARNING: REMOTE HOST IDENTIFICATION HAS CHANGED!. This example could be a deliberate attempt at a man-in-the-middle attack, but a changed hostkey often has a more harmless explanation: The administrator changed the key or reinstalled the system. If you are sure that the explanation is harmless, you can launch a text editor, open the `~/.ssh/known_hosts` file, and delete the entry in question.

Using Public-Key Authentication An alternative to standard password authentication to connect to a remote ssh server is to use public key authentication. With this approach, a user creates a public-private key pair on their local machine, and then distributes the public key to all the machines he wants to connect to. When an SSH client attempts to connect to one of these machines, the key pair will be used to authenticate the user, instead of the user being prompted for his password.

A key pair is generated with the `ssh-keygen` command. One of the decisions to make when creating a key pair is whether it will have a passphrase or not. If you create a key with a passphrase, you will be prompted for this passphrase every time you attempt to connect to the server you copied the key to. This will not appear much different from password authentication, though it allows you to use the same passphrase to login to multiple remote machines rather than having to remember different passwords for each of them. If you choose to create a key without a passphrase, you will be logged in to a remote server without having to type a password or passphrase. This provides extreme ease of use. However, it also means that should anyone get ahold of your private key, they would be able to login to any machine where you set the public key up. OpenSSH allows to generate a new identity key pair (RSA or DSA keys) as an ordinary unprivileged user, and store it in your `~/.ssh` directory on your local workstation.

```
$ ssh-keygen -t rsa
```

Distributing the public key is accomplished by appending it to the `~/.ssh/authorized_keys` file in the user's directory on each remote machine. This can be accomplished (via password authentication) using this ssh command:

```
$ cat id_rsa.pub | ssh username@hostname 'umask 077; cat >> .ssh/authorized_key'
```

(your key may be in the file `id_dsa.pub` instead). Note that the `umask 077` is in the command because SSH requires that this file be readable only by the owner.

You can also copy your new public key (`id_rsa.pub`) using the `ssh-copy-id` command:

```
$ ssh-copy-id -i id_rsa.pub user1@192.168.0.1
```

`ssh-copy-id` copies identity keys in the correct format, makes sure that file permissions and ownership are correct, and ensures you do not copy a private key by mistake.

Key-based authentication is good for brute-force protection, but entering your key passphrase at each and every new connection does not make your life any easier than using regular passwords.

A more secure but still relatively easy to use alternative, is to create your key pair with a passphrase, but then set up the `ssh-agent` program, which can automatically supply your passphrase when requested. Once `ssh-agent` is running, you use the `ssh-add` program to add your key(s). It is at this point that you are prompted for your passphrase(s). From this point on in the session, whenever an SSH client would need a passphrase, the agent supplies it. Normally the `ssh-agent` will be set up to start automatically on login. This program is a session-based daemon that caches the current user's decrypted private key. When the ssh client starts running, it tries to identify a running

instance of ssh-agent that may already hold a decrypted private key and uses it, instead of asking again for the key's passphrase. When starting the first instance of ssh-agent on your system, you have to manually add the keys that ssh-agent should keep in its cache. Starting ssh-agent usually happens in one of the session startup files such as ~/.bash_profile. This means a new instance of ssh-agent starts and asks for your key passphrase every time a new terminal session is started. This inconvenience prompted the creation of keychain, an ssh-agent wrapper that looks for a prior running instance before starting ssh-agent.

keychain improves on ssh-agent by checking for a running ssh-agent process when you log in and either hooking into one if it exists or starting a new one up if necessary (Some X GUI environments also have something similar available.) To set up keychain, first install the keychain and ssh-askpass packages for your system. For Debian/Ubuntu, use this:

```
$ sudo apt-get install keychain ssh-askpass
```

Now edit your ~/.bash_profile. file to include these lines:

```
#!/bin/bash
/usr/bin/keychain ~/.ssh/id_rsa ~/.ssh/id_dsa > /dev/null
source $HOME/.keychain/$HOSTNAME-sh
```

In line 02, you should include the paths to any keys that you want keychain to manage; check your ~/.ssh directory. Route the output to /dev/null to avoid getting the agent information output to the screen every time you start a new terminal. Line 03 sources ~/.keyring/$HOSTNAME-sh(which is created when keychain first runs to store ssh-agent information) to access SSH_AUTH_S0CK, which records the Unix socket that will be used to talk to ssh-agent. The first time the keychain process is started up (so, the first time you log in after setting this up), you'll be asked to enter the passphrases for any keys that you've listed on line 02. These will then be passed into ssh-agent. Once you have a bash prompt, try logging into a suitable machine (that is, one where you have this key set up) with one of these keys. You won't need to use your passphrase.

Now, if log out, and then log back in again, you shouldn't be challenged for your passphrases, because the ssh-agent session will still be running. Again, you should be able to log in to that suitable machine without typing your passphrase. The ssh-agent session should keep running until the next time you reboot.

This long-term running may be seen as a security issue. If someone managed to log in as you, they'd automatically be able to use your keys, as well. If you use the –clear option for keychain, then every time you log in, your passphrases will be cleared, and you will be asked for them again. Substitute this line for the earlier line 02:

```
02 /usr/bin/keychain --clear ~/.ssh/id_rsa ~/.ssh/id_dsa > /dev/null
```

The passphrases are cleared when you log in, not when you log out. This feature means that you can set your cronjobs, and so on, to get the passphrases via ssh-agent, and it won't matter if you've logged out overnight.

SSH even understands how to forward passphrase requests back through a chain of connections, to ssh-agent running on the ultimate originating machine (i.e., if you ssh from localhost to remote1 and from there ssh to remote2, the passphrase request can be sent back to localhost). This is known as agent forwarding, and it must either be enabled in the client configuration files on the machines, or the A option to ssh must be used when each connection is started.

Executing remote commands The main purpose of SSH is to execute commands remotely. As we have already seen, immediately after a successful SSH log-in, we're provided with the remote user's shell prompt from where we can execute all sorts of commands that the remote user is allowed to use. This 'pseudo' terminal session will exist as long as you're logged in. It is also possible to execute commands on a one-at-a-time basis without assigning a pseudo-terminal, as follows:

```
$ ssh user@192.168.0.1  'uname -a'
user@192.168.0.1's password:
Linux 192.168.0.1 2.6.28-9-generic #31-Ubuntu SMP Mon Jul 01 15:43:58 UTC
2013 i686 GNU/Linux
```

Thus, the ssh client can be used to execute a single command line on a remote machine. The basic format for a command to do this is:

```
ssh username@hostname 'COMMAND_LINE'
```

COMMAND_LINE is any legal Bash command line, which may contain multiple simple commands separated by ";" or pipelines, etc. Note that unless COMMAND_LINE is a simple command, it is good to quote it as shown.

For example, to count the number of files in username's Documents directory on hostname:

```
ssh username@hostname 'ls -1 Documents | wc -l'
```

or

```
ssh username@hostname 'ls -1 Documents' | wc -l
```

The difference between these examples is that the first command does the counting remotely, while the second does it locally. One of the most versatile aspects of the remote command capability is that you can use it in conjunction with piping and redirection.

Running XWindow applications remotely Another interesting feature is that SSH can do X11 Session Forwarding. To enable this feature you have to use the -X parameter[1] of the client. This parameter provides a form of tunneling, meaning "forward the X connection through the SSH connection". Example:

```
$ ssh -X user1@192.168.0.1
```

If this doesn't work, you may have to setup the SSH daemon (sshd) on the remote host to allow X11Forwarding. Check that the following lines are set in the /etc/ssh/sshd_config file on that host:

```
X11Forwarding yes
X11DisplayOffset 10
X11UseLocalhost yes
```

If you modify the sshd configuration file, you will need to restart the SSH service. You can to this typing the following command:

```
# service ssh restart
```

Port forwarding On the other hand, like X11 session forwarding, SSH can also forward other TCP ports, both, forward and backwards across the SSH session that you establish. As an illustrative example, let us consider the scenario of Figure 28.7. In this example, we want to use the TCP port 8080 in our localhost to send traffic to an apache WEB server that is running on 192.168.0.3:80. To do so, we are going to use a SSH tunnel through the host 192.168.0.2. The command to accomplish this is:

```
$ ssh -L 8080:192.168.0.3:80 user1@192.168.0.2
```

The -L parameter, which means Local port, can be interpreted as:

```
<tunnel-listening-port>:<connect-to-host>:<connect-to-port>
```

Once the forward SSH tunnel is established, it works as follows:

- When any process creates a connection to the port <tunnel-listening-port> in the client's host (8080 in our example), ssh sends the received data through the SSH connection with the sshd server (192.168.0.2:22).
- Then, the server sshd establishes a new TCP connection to <connect-to-host>:<connect-to-port> (192.168.0.3:80) using a new client port (11300).
- Notice that from the point of view of <connect-to-host> (192.168.0.3), its as if the connection came from the SSH server that you login to (192.168.0.2) and notice also that this second TCP connection is not encrypted.

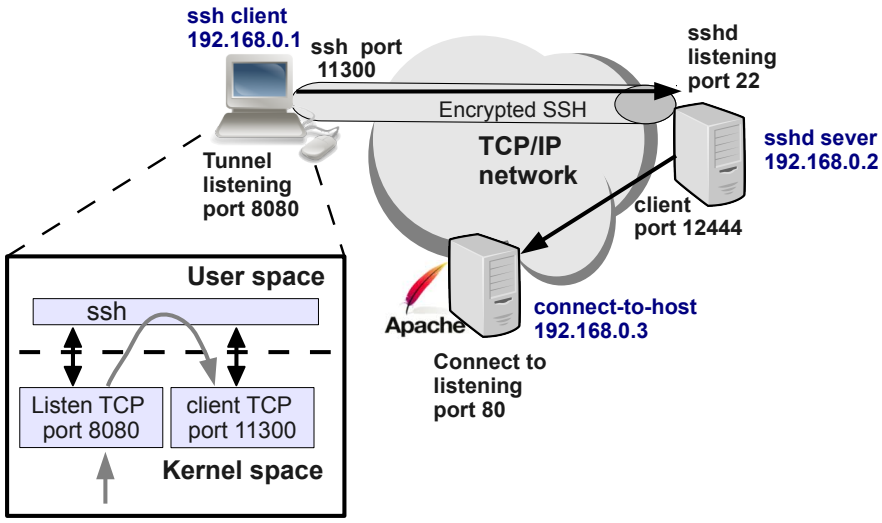


Figure 28.7: SSH port forwarding example.

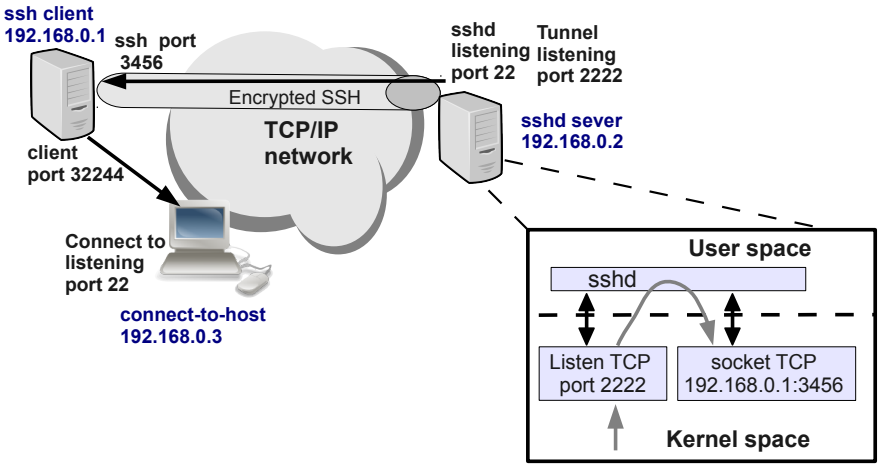


Figure 28.8: SSH port reverse forwarding example.

You can also create a reverse port forwarding (see Figure 28.8). To set up a reverse tunnel, you can type:

```
$ ssh -R 2222:192.168.0.3:22 192.168.0.2
```

The -R parameter, which means Reverse port, can be interpreted as:

```
<tunnel-listening-port>:<connect-to-host>:<connect-to-port>
```

Once the reverse SSH tunnel is established, it works as follows:

- When any process creates a connection to the port `<tunnel-listening-port>` (2222 in our example) in the SSH server, `sshd` sends the received data through the SSH connection back to the `ssh` client (192.168.0.1:11300).
- Then, the client `ssh` establishes a new TCP connection using a new client port (32244) to `<connect-to-host>:<connect-to-port>` (192.168.0.3:22).

[1]For some newer programs and newer versions of X windows, you may need to use the -Y option instead for trusted X11 forwarding.

- Again, from the point of view of <connect-to-host> (192.168.0.3), it is as if the connection came from the SSH client (192.168.0.1) and this second TCP connection is not encrypted.

We must point out that the <tunnel-listening-port> is only available to local processes in a SSH tunnel. Another way to express this is to say that these ports are bound to the local loopback interface only.

Finally, we have some other useful options for setting SSH tunnels. The "-n" option tells `ssh` to associate standard input with /dev/null, "-N" tells `ssh` to just set up the tunnel and not to prepare a command stream, and "-T" tells `ssh` not to allocate a pseudo-tty on the remote system. These options are useful for tunnels because, unlike a normal SSH login session, usually through a tunnel no actual commands are sent. Thus, we can use the following command to set up our previous reverse tunnel using these parameters:

```
$ ssh -nNT -R 2222:192.168.0.3:22 192.168.0.2
```

28.4 Secure Copy and File transfer

SSH also offers the file transfer facility between machines on the network and is highly secure, with SSH being an encrypted protocol. Also, the transfer speed can be improved by enabling compression. Two significant data transfer utilities that use the SSH protocol are SCP and SFTP. SCP stands for Secure Copy. We can use it to copy files from a local machine to a remote machine, a remote machine to a local machine, and a remote machine to another remote machine.

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 your local host to a remote host or vice versa. For example, the basic command that copies a file called file.txt from the local host to a file by the same name on a remote host is:

```
scp file.txt username@remotehost:
```

Note how the lack of a destination filename just preserves the original name of the file. This is also the case if the remote destination includes the path to a directory on the remote host. To copy the file back from the server, you just reverse the from and to:

```
$ scp username@remotehost:file.txt file.txt
```

If you want to specify a new name for the file on the remote host, simply give the name after the colon on the to side:

```
scp file.txt username@remotehost:anothername.txt
```

If you want to copy a file to a directory relative to the home directory for the remote user specified, you can type:

```
scp file.txt username@remotehost:mydirectory/anothername.txt
```

You can also use absolute paths, which are preceded with a "/". To copy a whole directory recursively to a remote location, 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:
```

Sometimes you will want to preserve the timestamps of the files and directories and if possible, the users, groups and permissions. To do this, use the "-p" option.

```
scp -rp Documents username@remotehost:
```

SFTP stands for Secure File Transfer Protocol. It is a secure implementation of the traditional FTP protocol with SSH as the backend. 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@s192.168.0.2'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>
```

If the port for the target SSH daemon is different from the default port, we can provide the port number explicitly as an option, i.e., `-oPort=port_number`. Some of the commands available under `sftp` are:

- `cd`—to change the current directory on the remote machine
- `lcd` —to change the current directory on localhost
- `ls`—to list the remote directory contents
- `lls`—to list the local directory contents
- `put`—to send/upload files to the remote machine from the current working directory of the localhost
- `get`—to receive/download files from the remote machine to the current working directory of the localhost
- `sftp` also supports wildcards for choosing files based on patterns.

28.5 Practices

28.6 Practices

Exercise 28.1–

Start the scenario *basic-netapps* on your host platform by typing the following command:

```
host$ simctl basic-netapps start
```

This scenario has two virtual machines `virt1` and `virt2`. Each virtual machine has two consoles (0 and 1) enabled. E.g. to get the console 0 of `virt1` type:

```
host$ simctl basic-netapps get virt1 0
```

1. Examine your `~/.ssh` directory. Remove the file known_hosts `virt1`.
2. Capture in `tap0` and open an SSH session from `virt2` to `virt1` with the command:

   ```
   virt2.0# ssh 10.1.1.1
   ```

What is the message you get? Note the key fingerprint of the machine before accepting the connection. What is the difference between the fingerprint and the public key? Why did ssh give you the fingerprint? Could this be a "man in the middle" attack?

3. Examine the output for the following steps:
   - found host key
   - what type of authentications will be tried
   - where the public key was looked for
   - when the password was tried
   - when the password was accepted and the interactive session was begun.
4. On your `virt2`, look into the directory `~/.ssh` directory. Which files are there and what is their content? What effect do they have on connecting to `virt1`?
5. Close the ssh session and start it again. What is the message you get now? Why?
6. Close the ssh session. Modify the corresponding files in `virt2` to recreate a"man in the middle" attack. Which file did you modified?

Exercise 28.2–

1. Capture in `tap0` and open an SSH session from `virt2` to `virt1` with the command:

   ```
   virt2.0# ssh 10.1.1.1
   ```

2. Which is the purpose of the first two SSH packets?
3. Explain the content of client Key Exchange packet and the content of server Key Exchange packet?
4. Does your trace contains Diffie–Hellman key exchange packets? What are the purpose of these packets?
5. Are the payload and the message authentication code (MAC) encrypted together or separately?
6. Using the `time` command and `openssl`, write a simple script to find out which is the fastest encryption algorithm Configure the ssh service to use this algorithm. Which files did you modified?

Exercise 28.3– So far to log in a remote machine using SSH we have used a password as authentication method. Now let's set up user accounts so you can log in using SSH keys.

1. Generate both a DSA and a RSA key pair. Make sure you use a passphrase. Which files have been created? What is their purpose? Are the permssions configured correctly? Where is the passphrase stored?
2. open an SSH session from `virt2` to `virt1` with the command:

   ```
   virt2.0# ssh 10.1.1.1
   ```

   At this point, which authentication method is used? Why?
3. Modify the corresponding files to authenticate using the DSA key.
4. Once you have successfully configured the private key authentication public, you will need to enter the passphrase each time you open an SSH session to `virt1`. Configure the `ssh-agent` to avoid entering the passphrase each time. What commands did you have to use and that machine?

5. Close the ssh session and the terminal where the ssh-agent was running? Open a new session, do you need to enter the passphare again? How could you solve that?

Exercise 28.4–

In this exercise set you will practice transferring files between systems using the secure copy (`scp`) and secure ftp (`sftp`) commands. Although both `scp` and `sftp` can be used to transfer files, each has its own strengths and weaknesses.

1. Create a text file in `virt2` and copy it to `virt1` using both `sftp` and `scp`. Specify the commands used. Can you transfer a file using a `sftp` in a single command line? If so, specify the command.

2. Create another text file in `virt2` in your home directory. Using a single `sftp` command create a `~/tmp/` directory in `virt1`, copy there the file and transfer it back to `virt2` and move it to the desktop.

Exercise 28.5–

In this practice, we are going to examine two ways of using an SSH session to establish X connections between a graphical application and an X server. For this purpose, we are going to use two Linux machines: your physical host and a virtual guest. Both machines will be connected with an `uml_swith`. Then, we will use the `xeyes` graphical application in the guest to establish an X connection to the X server of the host to view the "eyes". You have to try the two following ways of doing this:

```
host$ ssh -X user@192.168.0.2
```

or:

```
host$ xhost +192.168.0.2
host$ ssh user@192.168.0.2
guest$ export DISPLAY=192.168.0.1:0.0
guest$ xeyes
```

1. Explain in detail the differences between these two ways of establishing the X connection between the `xeyes` running in the guest and the X server running in the host. Use `netstat` and `wireshark` to provide a detailed explanation about how the different TCP connections are established.

2. Explain why when you use the first way (with the -X option of SSH) the command does not require the `xhost` command.

Part VI

Appendices

Appendix A

Ubuntu in a Pen-drive

Ubuntu in a Pen-drive

Install

In this appendix we show how to install Unbuntu 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)

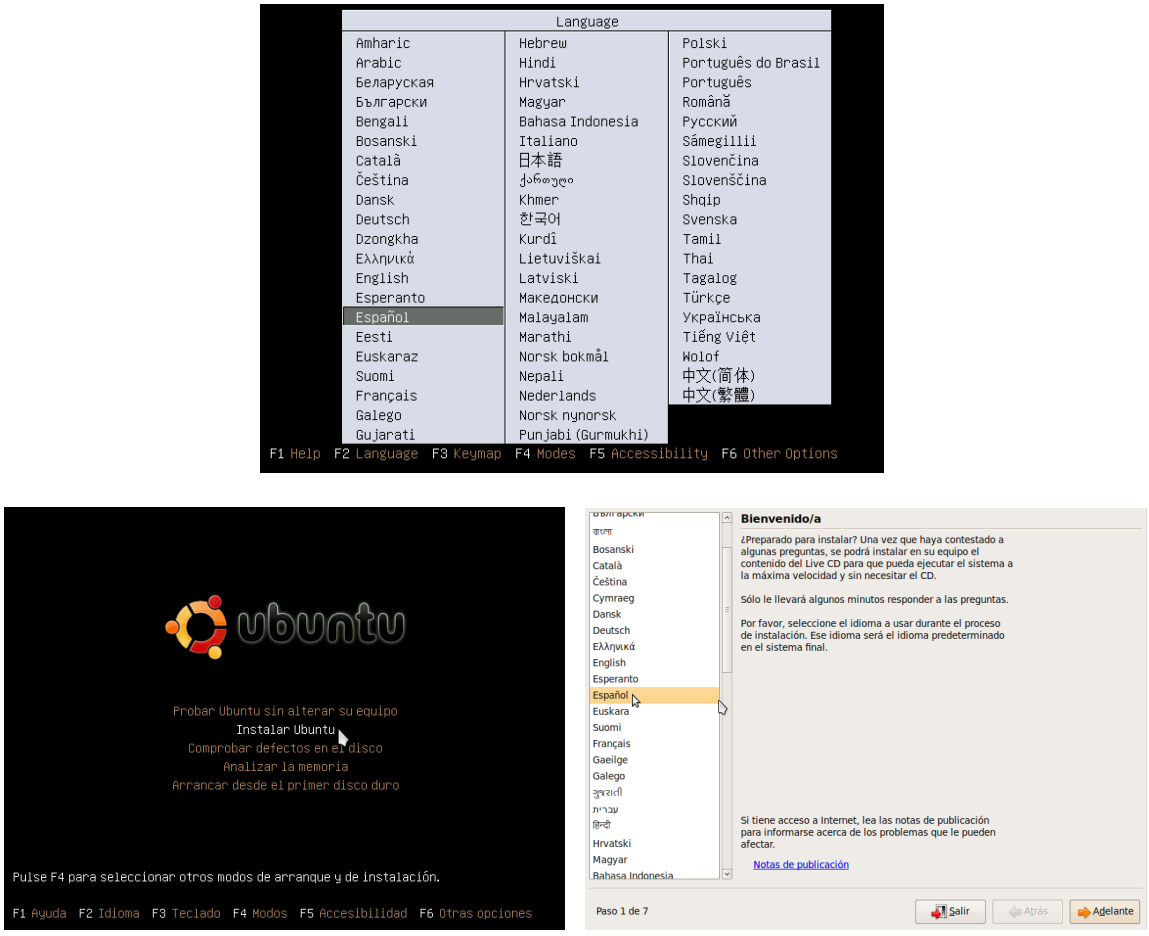


Figure A.1: First selection windows

Then, we select the time zone and the keyboard layout (Figure A.2).

Figure A.2: Time zone and Keyboard layout

Now, we must select the disk and the partition in which we want to install the OS.

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.3:

Now, we must select to specify disk partitions manually (advanced).

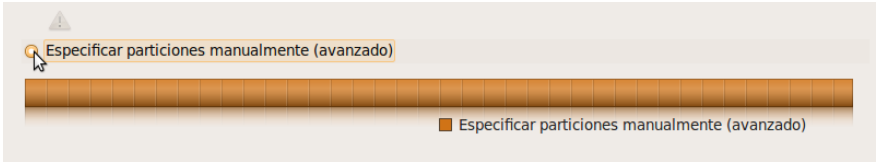


Figure A.4: Advanced specification of disk partitions

We must properly identify the disk device (view section 6.13 in Chapter **??**).

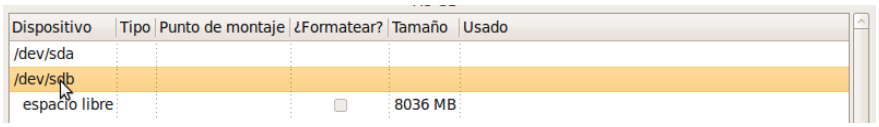


Figure A.5:

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.6). **Note. This account will be by default in the "sudoers" file with administration capabilities.**

Figure A.6: 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.7: 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.

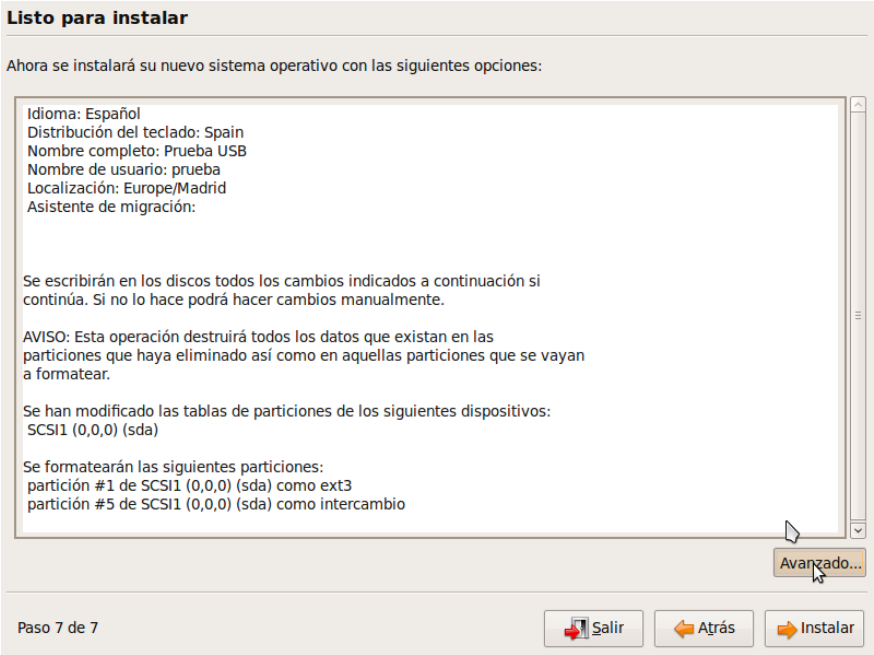


Figure A.8: Final installation step

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.

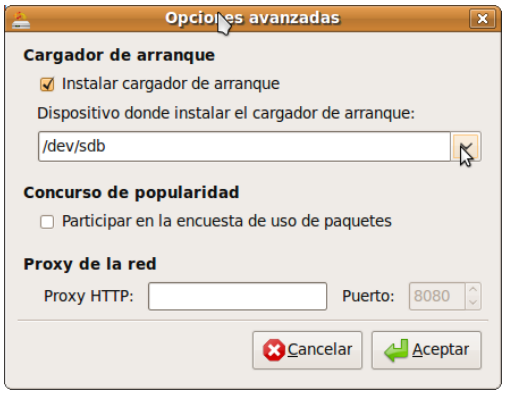


Figure A.9: Selecting the partition of the Boot loader

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.

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 in /etc/fstab.

- 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,noatime          0   0
  tmpfs      /var/tmp      tmpfs      defaults,noatime          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.

- 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.10). When we reboot the system all the previous changes must be active.

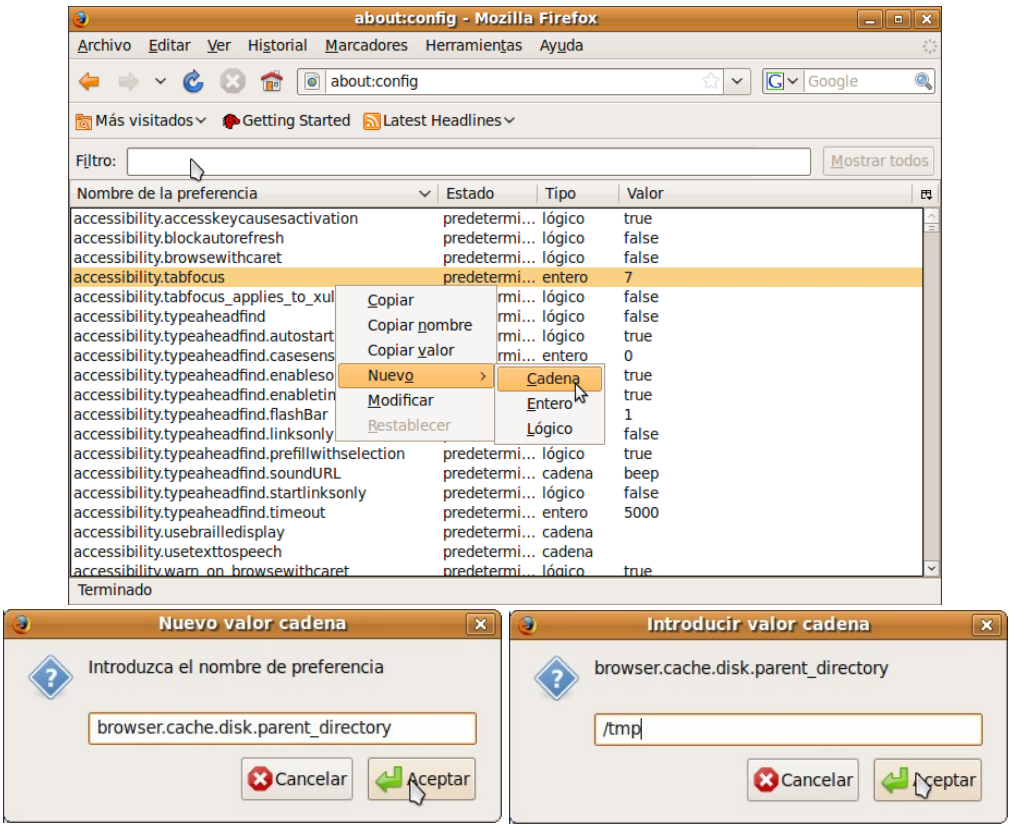


Figure A.10: Firefox cache in /tmp

Appendix B

Answers to Practices

Exercise 4.1

1. To open the manual of `ps` type: `man ps`
   Search and count with /ppid and n for next.
2. Type: `ps -o pid,tname,cmd`
3. The option *comm* shows the command without parameters, while *cmd* shows the command line parameters.
4. The father is a bash, the grandfather is a gnome-terminal and finally these are children of init.
5. The new windows or tabs are not new gnome-terminals, they are the same process but we have new bash processes.
6. Each xterm terminal is a new process connected with its corresponding bash.
7. The xeyes and the xclock are executed. The former is in foreground, while the latter is in background.
8. `t3$ ps -Ao pid,ppid,state,command`
   To view the tty type: `t3$ tty` or add the column tname to the `ps` command
9. Type: `t3$ kill -TERM PID`
10. Now xclock is an "orphan" process and its father is init.
11. `$ kill -9 PID` or `$ kill -KILL PID`
12. `t1$ xclock`
13. `t1$ xclock`
    When a process is stopped and continued it continues running in background.
14. `t1$ xclock`
    CRL+z, bg, kill %JOB
15. More job control...
16. `$ ps; sleep 3; ps`

17. `$ ps -ñ || ps`

18. `$ sleep || sleep || ls`

    The ls command is executed and the two sleep commands give an error sleep: missing operand Try ‘sleep –help’ for more information.

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

    first sleep error, second sleep not executed, ls and ps executed.

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

    first sleep error, second sleep not executed, ls executed and ps not executed.

Exercise 4.2

1. `$ nice -n 18 xeyes & $ ps -o cmd,pid,ni`
2.
```
#!/bin/bash
echo Type the number to be multiplied by 7:
read VAR
echo The multiplication of $VAR by 7 is $[VAR*7].
```

3.
```
#!/bin/bash
trap "echo waiting operand" USR1
echo Type the number to be multiplied by 7:
read VAR
echo The multiplication of $VAR by 7 is $[VAR*7].
```

   To figure out the correct PID to which send the kill, you can lauch the script in background. Then, you can return the script execution to foreground.

Exercise 6.1

1. `$ cd ../../etc`
2. `$ cd /home/XXXX`
3. `$ cp ../../etc/passwd .`
4. `$ mkdir dirA1; mkdir dirA2; ...`
5. `$ rmdir dir[A,B][1,2]`
   or
   `$ rmdir dir[A,B]?`
6. `$ rmdir dirC?`
7. `$ touch temp`

8. `$ cat temp`

   You can also use less or more

9. `$ stat temp`

10. Regular, empty file.

11. Your can't because you don't have permissions.

12. `$ chmod u-x without_permission`

13. You can't copy to the without_permission directory

14.

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

Exercise 6.2

1. `$ touch orig.txt`
   `$ ln -s orig.txt link.txt`

2. `$ vi orig.txt`

3. We observe the same contents as the orig.txt file.

4. The results are the same.

5. You cannot do it.

6. You cannot take away the permissions of a symbolic link.

7. `$ cat link.txt`
   `cat: link.txt: No such file or directory`

   When editing a new file orig.txt is created

8. Each file has just one link.

9. `$ ln orig.txt hard.txt`
   `$ stat hard.txt`

   Each file has two links. In fact, they are the same file.

10. This is like renaming the file: `mv orig.txt hard.txt`

11. `$ grep http /etc/services`
    `$ grep HTTP /etc/services`
    or
    `$ grep -i http /etc/services`

12. `$ cut -d ``:'' -f 1,4-9 /etc/group`

13. `$ file text1.txt`
    `text1.txt: UTF-8 Unicode text`

14. ñ in UTF8 is 0xc3b1 a=0x61 b=0x62 (both ISO and UTF8)

15. 0x0a is LF

16. 0x0d is CR

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

17. Unix (and new Mac): LF

    Windows: CR+LF

    Classical Mac: CR

18. ñ=f1

Exercise 8.1

1.
```
$ ls -R > mylist.txt
$ echo "CONTENTS OF ETC" >> mylist.txt
$ tail -10 mylist.txt
```

2.
```
$ echo "CONTENTS OF ETC" > aux1.txt
$ cat aux1.txt mylist.txt > aux2.txt
$ mv aux2.txt mylist.txt
$ rm aux1.txt
$ head -10
```

3.
```
$ ls /bin | wc -l
```

4.
```
$ ls /bin | head -n 3
$ ls /bin | head -n 3 | sort -r
```

5.
```
$ ls /bin | sort -r | tail -n 3
$ ls /bin | sort -r | tail -n 3 | sort -r
```

6.
```
$ cat /etc/passwd /etc/group | wc -l
```

7.
```
$ ps -Ao ppid,pid,cmd | grep /sbin/init | grep -v grep
```

Exercise 8.2

1.
```
$ cat /etc/passwd > /dev/pts/X
```

   We can find the tty (X) of the other terminal typing the `tty` command in the terminal in which the file is going to be displayed.

2. Assuming that the first terminal (t1) is in /dev/pts/1 and the second terminal (t2) is in /dev/pts/2. Then:

```
t1$ cat - > /dev/pts/2
t2$ cat - > /dev/pts/1
```

Note. This also works without the dash in `cat`, as this command reads from stdin by default.

Exercise 8.3

With these commands we create a loop, so CPU usage is high.

Exercise 8.4

1. ```
   t1$ tail -f re.txt | grep --line-buffered kernel | \
   > cut -c 1-10
   ```
2. ```
   t1$ tail -f re.txt | grep --line-buffered \
   > ' [^aeiou]*[aeiou][^aeiou]*[aeiou][^aeiou]*[aeiou][^aeiou]*$'
   ```

Exercise 8.5

1. ```
   t1$ less /etc/passwd
   t2$ less /etc/passwd
   t3$ lsof | grep /etc/passwd
   ```
2. ```
   $ fuser -k /etc/passwd
   ```
3. ```
   t1$ touch file.txt
   t1$ exec 4< file.txt
   t1$ exec >file.txt
   t1$ ./openfilescript.sh
   t2$ cat file.txt
   COMMAND    PID    USER   FD   TYPE DEVICE SIZE/OFF     NODE NAME
   script.sh 5588 user1    0u   CHR  136,3      0t0        6 /dev/pts/3
   script.sh 5588 user1    1w   REG    8,6        0 10886559 /home/user1/file.txt
   script.sh 5588 user1    2u   CHR  136,3      0t0        6 /dev/pts/3
   script.sh 5588 user1    4r   REG    8,6        0 10886559 /home/user1/file.txt
   Hello!!
   COMMAND    PID    USER   FD   TYPE DEVICE SIZE/OFF     NODE NAME
   ```
4. You see nothing in the output and nothing in file.txt. This is because the open file descriptor of the bash of the first pseudo-terminal has open files that currently do not exist.

Exercise 20.1

Exercise 20.2

Exercise 20.3
   ```

Exercise 20.4

Exercise 20.5

Exercise 20.6

En la transfer binaria hay conversion y desconversion LF a CRLF. El cliente linux lo deja como el original.

Exercise 22.1

This script provides information about the script. It is self-explanatory.

Exercise 22.2

This script removes temporary files from our working directory. It asks the user to make sure about this and uses a case to perform the correct action. It also manages three different exit status.

Exercise 22.3

```
#!/bin/bash
# AUTHOR: teacher
# DATE: 4/10/2011
# NAME: hypotenuse2.sh
# SYNOPSIS: procedure.sh arg1
# DESCRIPTION: Calculates the squared hypotenuse of a rectangular triangle.
# HISTORY: First version

hyp2(){
  local c1 c2
  c1=$1; c2=$2
  ((c1*=c1))
  ((c2*=c2))
  echo "The squared hypotenuse of cathetus $1 and $2 is: $((c1+c2))"
}

if [ $# -ne 2 ]; then
   echo "Usage: $0 c1 c2"
   exit 1
fi
hyp2 $1 $2
```

Exercise 22.4

We find out if the character hexchar is printable. If the character is not printable, we return an error code. Printable characters are those of the ASCII code 0x21-0x7F (33-127).

Exercise 22.5

This script searches recursively a file from a working directory.

```
#!/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
  # Enter in the working directory
  [ $DEBUG -eq 1 ] && echo "Entrant a $1"
  cd $1
  # For each file in the current directory we have to figure out
  # if the file is a directory or if it is a regular file.
  # If the file is a directory,
  # we call the function with the found directory as argument.
  # If the file is a regular file,
  # we look if the file is the file we were looking for.
  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"
  # We exit the inspected directory
  cd ..
}

# main
# We define a global variable that manages the printing
# of "debug" messages.
# If DEBUG=1, then we print debug messages
DEBUG=0

# First, we verify that the invocation of
# the script is correct.
if [ $# -ne 1 ]; then
  echo "Usage: $0 file_to_search"
  exit 1
fi

# We use a global variable to store
# file name we're searching for.
FILE_IN_SEARCH=$1

# We call the function using the current directory as argument.
```

```
# The search will take place from the current directory to the children directories.

search_in_dir `pwd`
```

Exercise 22.6

```
#!/bin/bash
# AUTHOR: teacher
# DATE: 4/10/2011
# NAME: factorial.sh
# SYNOPSIS: factorial.sh arg
# DESCRIPTION: Calculates the factorial of arg
# HISTORY: First version

factorial() {
  local I AUX H
  I=$1
  H=$1
  if [ $I -eq 1 ]; then
    echo 1
  else
    ((I--))
    AUX=`factorial $I`
    echo $((H*AUX))
  fi
}

if [ $# -eq 1 ]; then
  echo The factorial of $1 is: $(factorial $1)
  exit 0
else
  echo "Usage: $0 integer_num"
  exit 1
fi
```

Exercise 24.1

1. `host$ uml-kernel ubda=filesystem.fs mem=128M`
   In our case, the Kernel versions of the guest and host are different.
2. `guest# id`
   `guest# who`
3. `guest# useradd -m -s /bin/bash user1`
   `guest# useradd -m -s /bin/bash user2`
   `guest# useradd -m -s /bin/bash user3`
4. `guest# su user1`
   `guest# id`

uid is in the /etc/passwd file

gid is in the /etc/group file

5. *user1* does not appear as logged in:

```
user1@guest:~$ who
root     tty0         Sep  3 02:30
```

First we need to provide a password with the `passwd` command for *user1*, then exit and then login again as *user1*.

```
user1@vm:~$ who
user1@vm:~$ last
user1@vm:~$ id
```

6.
```
user1@guest:~$ su
guest# groupadd someusers
guest# cat /etc/group
guest# usermod -G someusers -a user1
guest# usermod -G someusers -a user2
guest# usermod -G someusers -a user3
guest# exit
user1@guest:~$ id
uid=1000(user1) gid=1000(user1) groups=1000(user1)
```

As shown by the id command, the *user1* still does not belong to the group someusers. As you see, these changes are not effective over currently logged users.

7. When you switch to *user1* you now see that now the user belongs to the group "someusers".

```
guest# groupadd otherusers
guest# usermod -G otherusers -a user2
guest# usermod -G otherusers -a user3
```

8.
```
host$ dd bs=1M if=/dev/zero of=second-disk.fs count=30
host$ mkfs.ext3 second-disk.fs -F
host$ ./uml-kernel ubda=filesystem.fs ubdb=second-disk.fs mem=128M
```

9.
```
guest# mkdir /mnt/extra
guest# mount /dev/ubdb /mnt/extra
guest# mount
guest# df -h
guest# mkdir /mnt/extra/etc ; mkdir /mnt/extra/home
```

10. `/etc/fstab:`

```
proc         /proc          proc     defaults   0 0
devpts       /dev/pts       devpts   mode=0622  0 0
/dev/ubda    /              ext3     defaults   0 1
/dev/ubdb    /mnt/extra     ext3     defaults   0 1
```

11.
```
host# mkdir mnt
host# mount -o loop second-disk.fs ./mnt
host# cp /etc/passwd ./mnt/etc
host# umount ./mnt
host$ ./uml-kernel ubda=filesystem.fs ubdb=second-disk.fs mem=128M
```

```
guest#  mount
/dev/ubda on / type ext3 (rw)
/dev/ubdb on /mnt/extra type ext3 (rw)
guest# ls /mnt/extra/etc/
passwd
```

12. `guest# dd bs=1M if=/dev/zero of=pen-drive.fs count=10`
```
guest# mkdir /mnt/usb
guest# mount -o loop pen-drive.fs /mnt/usb
guest# mkdir /mnt/usb/etc
guest# umount /mnt/usb
guest# mount -o loop pen-drive.fs /mnt/extra
guest# ls /mnt/extra/etc
guest# cp /etc/passwd /mnt/extra/etc
```
/mnt/extra/etc was empty when we mounted pen-drive.fs because this filesystem did not contain this file at this time. Then, we copied the /etc/passwd file of the guest to pen-drive.fs but after unmounting, we return to second-disk.fs so the file /mnt/extra/etc/passwd is obviously the original passwd file (the file that we copied from the host in the previous exercise).

13. We create root2 and mount /dev/ubda. As shown, a single device can be mounted several times in different mount points. Then, we change the root of the filesystem to /mnt/extra/root2 We still see with the `mount` command that /dev/ubda is mounted twice but we see nothing inside /mnt/extra. Finally, we get out of `chroot` with exit and unmount. **Note. `chroot` is used to change the root of the system. This allows for example, to use another filesystem containing a different linux distro (like a distro in a CD).**

14. `guest# chgrp someusers /mnt/extra`
```
guest# chmod g+rwx /mnt/extra
guest# su user1
user1@vm:~$ touch /mnt/extra/hello.txt
```

15. `guest# chgrp otherusers /mnt/extra`
```
guest# chmod 775 /mnt/extra
guest# su user1
user1@vm:~$ touch /mnt/extra/hello.txt
user1@vm:~$ exit
guest# su user3
user3@vm:~$ touch /mnt/extra/hello.txt
touch: cannot touch 'hello.txt': Permission denied
```

16. We show the configuration for *user1*:

```
guest# mkdir /mnt/extra/user1
guest# chown user1.someusers /mnt/extra/user1
guest# chmod 750 /mnt/extra/user1
```

17. links.sh:

```
#!/bin/bash
ln -s /mnt/extra /home/user1
ln -s /mnt/extra /home/user2
ln -s /mnt/extra /home/user3
```

To execute the script:

```
# chmod u+x
#./links.sh
```

18. ```
    # ln -s /mnt/extra /etc/skel
    # useradd -m -s /bin/bash user4
    # ls /home/user4 -l
    ```
    The link is created when you create the user.

19. ```
    $ logger "probing logging system"
    $ tail -10 /var/log/syslog
    ```

Exercise 28.5