Entorno de Programación

Tecnicatura Universitaria en Inteligencia Artificial

5 de marzo de 2025

Índice general

I Fundamentos 8

1 Historia 9
1.1 Orígenes de la computadora . . . 9
1.2 Avances de la guerra . . . 11
1.3 La informática moderna . . . 14
1.4 Primeras computadoras personales . . . 16
1.5 La era de la información . . . 18
1.6 Ejercicios . . . 21

2 Arquitectura de la computadora 22
2.1 Gabinete . . . 22
2.2 Placa madre . . . 23
2.3 Fuente de alimentación . . . 23
2.4 Microprocesador . . . 24
2.5 Memoria RAM . . . 25
2.6 Memoria secundaria . . . 25
2.7 Placa de video . . . 28
2.8 Teclado . . . 28
2.9 Ejercicios . . . 30

3 Representación de la información 31
3.1 Unidades de información . . . 31
3.2 Sistemas de numeración . . . 33
3.3 Codificación de caracteres . . . 35
3.4 Ejercicios . . . 39

4 Conceptos de programación 41
4.1 Introducción . . . 41
4.2 Lenguaje de programación . . . 42
4.3 Niveles de lenguajes . . . 42
4.4 Compiladores e intérpretes . . . 44
4.5 Otros conceptos . . . 45
4.6 Ejercicios . . . 46

5 Sistemas Operativos 47
5.1 Descripción . . . 47
5.2 Núcleo . . . 48

5.3 Proceso de arranque . . . 49
5.4 Interfaz de usuario . . . 51
5.5 Terminal . . . 53
5.6 Procesos . . . 55
5.7 Distribuciones . . . 56
5.8 Proceso de apagado . . . 58
5.9 Ejercicios . . . 59

6 Sistema de archivos 60
6.1 Acceso aleatorio y secuencial . . . 60
6.2 Descripción . . . 61
6.3 Particiones . . . 62
6.4 Estructura de directorios . . . 63
6.5 El sistema de archivos de Linux . . . 65
6.5.1 Esquema de particionado . . . 65
6.5.2 Nomenclatura de dispositivos . . . 66
6.5.3 Jerarquía de archivos . . . 67
6.5.4 Carpetas especiales . . . 69
6.5.5 Montaje . . . 69
6.5.6 Enlaces . . . 69
6.6 Ejercicios . . . 70

II Manejo de bash 71

7 Comandos básicos 72
7.1 Introducción . . . 72
7.1.1 `man` . . . 75
7.1.2 `clear` . . . 77
7.1.3 `echo` . . . 78
7.1.4 `history` . . . 78
7.1.5 Ejercicios . . . 79
7.2 Sistema de archivos . . . 80
7.2.1 `pwd` . . . 80
7.2.2 `cd` . . . 80
7.2.3 `ls` . . . 81
7.2.4 `tree` . . . 83
7.2.5 `mkdir` . . . 84
7.2.6 `rmdir` . . . 85
7.2.7 `touch` . . . 86
7.2.8 `rm` . . . 87
7.2.9 `cp` . . . 88
7.2.10 `mv` . . . 89
7.2.11 `df` . . . 89
7.2.12 `du` . . . 90
7.2.13 `ln` . . . 91

7.2.14 `lsblk` . . . 92
7.2.15 `mount` . . . 93
7.2.16 `umount` . . . 93
7.2.17 `find` . . . 94
7.2.18 Expansiones . . . 96
7.2.19 Ejercicios . . . 98
7.3 Contenido y filtros . . . 105
7.3.1 `nano` . . . 105
7.3.2 `cat` . . . 106
7.3.3 `less` . . . 107
7.3.4 `head` . . . 108
7.3.5 `tail` . . . 108
7.3.6 `sort` . . . 109
7.3.7 `uniq` . . . 110
7.3.8 `strings` . . . 110
7.3.9 `wc` . . . 111
7.3.10 `file` . . . 112
7.3.11 `cut` . . . 112
7.3.12 Expresiones regulares . . . 113
7.3.13 `tr` . . . 116
7.3.14 `grep` . . . 117
7.3.15 `sed` . . . 119
7.3.16 Ejercicios . . . 119
7.4 Secuenciación, redirección y tuberías . . . 123
7.4.1 Secuenciación . . . 123
7.4.2 Redirección . . . 124
7.4.3 Tuberías . . . 126
7.4.4 `tee` . . . 127
7.4.5 Ejercicios . . . 128

8 Comandos avanzados 130
8.1 Usuarios, grupos y permisos . . . 130
8.1.1 `whoami` . . . 130
8.1.2 `id` . . . 130
8.1.3 `who` . . . 131
8.1.4 `su` . . . 131
8.1.5 `sudo` . . . 132
8.1.6 `passwd` . . . 133
8.1.7 Permisos en Linux . . . 134
8.1.8 `chown` . . . 135
8.1.9 `chmod` . . . 136
8.1.10 `useradd` . . . 137
8.1.11 `userdel` . . . 138
8.1.12 `usermod` . . . 138
8.1.13 Ejercicios . . . 139
8.2 Procesos y tareas . . . 140

8.2.1 ps . . . 140
8.2.2 nice . . . 141
8.2.3 top . . . 141
8.2.4 Señales . . . 142
8.2.5 kill . . . 143
8.2.6 Tareas . . . 143
8.2.7 jobs . . . 144
8.2.8 bg . . . 144
8.2.9 fg . . . 145
8.2.10 disown . . . 145
8.2.11 wait . . . 145
8.2.12 exec . . . 146
8.2.13 Ejercicios . . . 146
8.3 Gestión . . . 147
8.3.1 apt . . . 147
8.3.2 free . . . 148
8.3.3 loadkeys . . . 148
8.3.4 setfont . . . 149
8.3.5 setxkbmap . . . 150
8.3.6 which . . . 150
8.3.7 startx . . . 150
8.3.8 Ejercicios . . . 151
8.4 Internet . . . 151
8.4.1 ping . . . 151
8.4.2 ssh . . . 152
8.4.3 scp . . . 153
8.4.4 wget . . . 154
8.4.5 curl . . . 154
8.4.6 w3m . . . 155
8.4.7 Ejercicios . . . 156
8.5 Servicios . . . 157
8.6 Logging . . . 157
8.7 Tareas programadas . . . 157
8.8 Compresión y backup . . . 157
8.8.1 tar . . . 157
8.8.2 gzip . . . 157
8.8.3 rsync . . . 158
8.8.4 dd . . . 159
8.8.5 Ejercicios . . . 159
8.9 Otros comandos . . . 159
8.9.1 alias . . . 159
8.9.2 unalias . . . 160
8.9.3 bc . . . 160
8.9.4 sha256sum . . . 161
8.9.5 time . . . 162
8.9.6 watch . . . 163

8.9.7 xclip . . . 164
8.9.8 xargs . . . 165
8.9.9 screen . . . 165
8.9.10 ranger . . . 165
8.9.11 Ejercicios . . . 165

9 Shell scripting 167
9.1 Introducción . . . 167
9.1.1 Comentarios . . . 167
9.1.2 Ejecución . . . 168
9.1.3 Ejercicios . . . 168
9.2 Variables . . . 169
9.2.1 Variables locales . . . 169
9.2.2 Variables de entorno . . . 170
9.2.3 Variables especiales . . . 171
9.2.4 Ejercicios . . . 172
9.3 Comandos . . . 172
9.3.1 test . . . 172
9.3.2 exit . . . 174
9.3.3 source . . . 174
9.3.4 read . . . 175
9.3.5 seq . . . 175
9.3.6 shift . . . 176
9.3.7 export . . . 176
9.3.8 trap . . . 177
9.3.9 Ejercicios . . . 177
9.4 Encomillados y escapes . . . 178
9.4.1 Carácter de escape . . . 178
9.4.2 Comillas simples . . . 179
9.4.3 Comillas dobles . . . 180
9.4.4 Encomillado ANSI-C . . . 181
9.4.5 Comillas invertidas . . . 182
9.4.6 Expresiones aritméticas . . . 182
9.4.7 Ejercicios . . . 183
9.5 Control de flujo . . . 183
9.5.1 if / else . . . 183
9.5.2 for . . . 185
9.5.3 while / until . . . 185
9.5.4 case . . . 186
9.5.5 select . . . 187
9.5.6 Funciones . . . 188
9.5.7 Ejercicios . . . 189

III Herramientas auxiliares 190

10 Control de versiones 191
10.1 Introducción . . . 191
10.1.1 Descripción . . . 191
10.1.2 Configuración . . . 193
10.1.3 Ejercicios . . . 194
10.2 Comandos básicos . . . 194
10.2.1 `init` . . . 194
10.2.2 `status` . . . 196
10.2.3 `add` . . . 197
10.2.4 `commit` . . . 199
10.2.5 `log` . . . 200
10.2.6 `restore` . . . 201
10.2.7 `diff` . . . 205
10.2.8 ~~`reset`~~ . . . 207
10.2.9 Ejercicios . . . 207
10.3 Trabajando con ramas . . . 208
10.3.1 Definición . . . 208
10.3.2 `branch` . . . 208
10.3.3 `switch` . . . 209
10.3.4 `merge` . . . 213
10.3.5 ~~`rebase`~~ . . . 214
10.3.6 ~~Resolución de conflictos~~ . . . 214
10.3.7 ~~Flujos de trabajos~~ . . . 214
10.3.8 Ejercicios . . . 214
10.4 Repositorios remotos . . . 215
10.4.1 `remote` . . . 215
10.4.2 `push` . . . 216
10.4.3 `fetch` . . . 218
10.4.4 `pull` . . . 221
10.4.5 `clone` . . . 221
10.4.6 ~~Fork~~ . . . 222
10.4.7 ~~Pull request~~ . . . 222
10.4.8 ~~LFS~~ . . . 222

11 ~~Contenedores~~ 223
11.1 ~~Simulación~~ . . . 223
11.2 ~~Emulación~~ . . . 223
11.3 ~~Virtualización~~ . . . 223
11.4 ~~Contenedores~~ . . . 223

12 ~~Otros conceptos~~ 224
12.1 ~~`venv`~~ . . . 224
12.2 ~~`chroot`~~ . . . 224
12.3 ~~Colab~~ . . . 224

Apéndice 225

13 Instalación de linux 226
13.1 Máquina virtual . . . 226
13.1.1 VirtualBox . . . 226
13.1.2 Lubuntu . . . 229
13.1.3 Guest Additions . . . 234
13.2 Instalación en pendrive . . . 234
13.3 Otras alternativas . . . 236
13.3.1 Subsistema de Windows para Linux . . . 236
13.3.2 Colab . . . 236
13.3.3 ~~Replit~~ . . . 237
13.3.4 ~~CoCalc~~ . . . 237
13.3.5 ~~Termux~~ . . . 237

14 Resolución de problemas 238
14.1 Activar *IVT/AMD-V* . . . 238
14.2 Librerías de C++ . . . 239
14.3 Poner `man` en español . . . 239
14.4 Actualizar Kernel para WSL . . . 240

15 ~~Programando un shell~~ 241
15.1 ~~Linea de comandos~~ . . . 241
15.2 ~~Variable PATH~~ . . . 242
15.3 ~~Manejo de argumentos~~ . . . 243
15.4 ~~Cambiar de directorio~~ . . . 244
15.5 ~~Prompt~~ . . . 245
15.6 ~~Alias~~ . . . 246
15.7 ~~Segundo plano~~ . . . 247
15.8 ~~Secuenciación~~ . . . 248
15.9 ~~Exit~~ . . . 249
15.10 ~~Historial~~ . . . 250

! Observación

Aquellos temas señalados en color gris, no serán evaluados *durante los exámenes*.

Parte I

Fundamentos

«Una computadora es para mí la herramienta más sorprendente que hayamos ideado. Es el equivalente a una bicicleta para nuestras mentes».

Steve Jobs

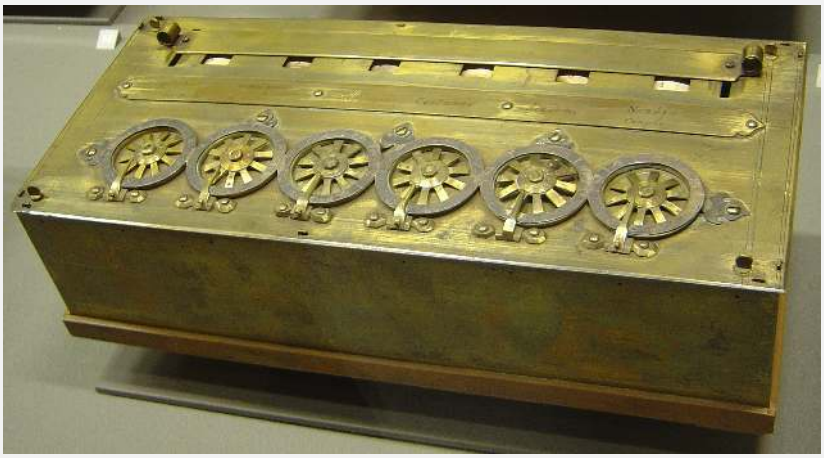
1

Historia

1.1 Orígenes de la computadora

Blaise Pascal había desarrollado en 1642 una de las primeras calculadoras mecánicas de la historia, conocida como la «*Pascalina*». Esta máquina, capaz de realizar sumas mediante el uso de engranajes y ruedas dentadas, fue un importante avance en el campo de la mecánica y la automatización, y sentó las bases para el desarrollo de las máquinas calculadoras modernas. La W *Pascalina* fue utilizada por matemáticos y científicos de la época, y su diseño influyó en el desarrollo de posteriores calculadoras mecánicas.

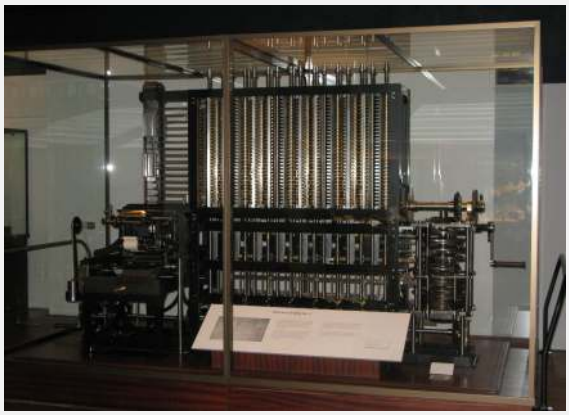
Figura 1.1: Pascalina



La «*Máquina Analítica*» fue un proyecto desarrollado por *Charles Babbage* en 1837, considerado como el primer prototipo de una computadora moderna. La W *Máquina Analítica* tenía la capacidad de almacenar programas y realizar operaciones matemáticas complejas. Es considerada el primer prototipo de sistema de procesamiento de datos automático.

Aunque Babbage nunca logró construir la máquina completa debido a problemas financieros y técnicos, su proyecto sentó las bases para el desarrollo de las computadoras modernas.

Figura 1.2: W *Máquina Diferencial*



El W telégrafo fue el primer sistema de comunicación eléctrica que permitió transmitir mensajes a larga distancia de manera rápida. Funcionaba mediante la codificación de letras y números en señales eléctricas, que luego eran enviadas a través de cables y decodificadas en el otro extremo. Fue inventado por *Samuel Morse* y patentado en 1837.

Figura 1.3: Telégrafo

El teletipo fue una invención que utilizaba un sistema de impresión automática para la transmisión de texto a través de una red telegráfica. Fue ampliamente utilizado durante la decada del 30, y su desarrollo permitió una comunicación más rápida y eficiente que el telégrafo.

Figura 1.4: W Teletipo Siemens t37h

! Observación

Las terminales de teletipo (TTY) se usaron también como terminales de computadoras en las primeras décadas de su historia.

«TTY» es un acrónimo de «TeleType».

1.2 Avances de la guerra

La computadora «Z3» , construida en 1941 por *Konrad Zuse*, era una máquina electromecánica basada en «W relés», que utilizaba un sistema binario para representar los datos y los cálculos. La W *Z3* contaba con una memoria programable, lo que permitía al usuario programar la máquina para llevar a cabo diferentes tareas.

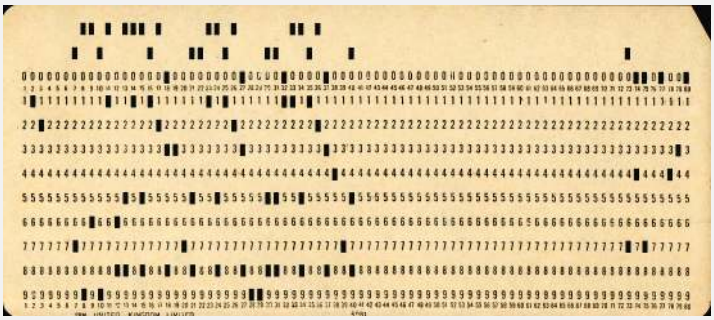
Figura 1.5: Réplica del Zuse Z3 exhibida en Múnich

La «▶ *Harvard Mark I*», una computadora electromecánica construida en 1944 por IBM y patrocinada por el gobierno de los Estados Unidos, fue diseñada por 👤 *Howard Aiken* y se encuentra en el Museo de la historia de la computación en la Universidad de Harvard.

Figura 1.6: W *Harvard Mark I*

Para leer los datos se utilizaban «▶ tarjetas perforadas», un medio de almacenamiento de datos utilizado históricamente en la informática. Estas tarjetas son cartulinas rectangulares que contienen información codificada mediante perforaciones ubicadas en filas y columnas.

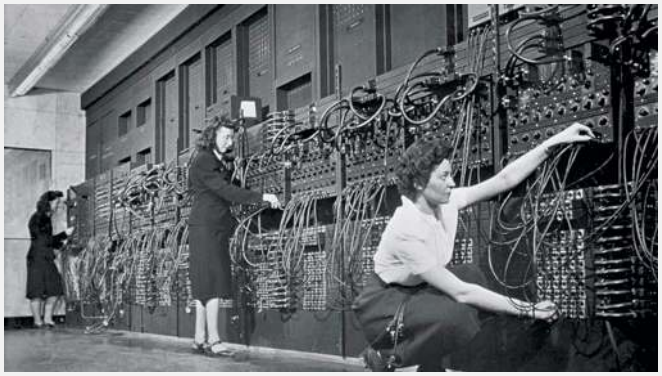
Figura 1.7: W Tarjeta perforada de 20 filas y 80 columnas.



La ▶ ENIAC fue una computadora electrónica programable construida por el gobierno de los Estados Unidos en 1946. Utilizaba tecnología de «W válvulas electrónicas» y era capaz de realizar cálculos numéricos complejos a alta velocidad.

«ENIAC» son las siglas de «Electronic Numerical Integrator And Computer».

Figura 1.8: W *Electronic Numerical Integrator And Computer*



Para programar la ENIAC se utilizaban «W *plugboards*» o paneles de conexiones. Estos eran paneles con enchufes y cables que permitían a los operadores configurar la función de la máquina de manera física, estableciendo conexiones entre diferentes componentes. Los operadores debían conectar cables en los enchufes correspondientes para definir la ruta de los datos y las operaciones que la máquina debía realizar.

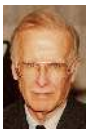
Figura 1.9: Panel de conexiones de la IBM 402.

! Observación

Aunque la tecnología de válvulas era superior a la de relés, el verdadero salto en el poder de cómputo se logró con la invención del W transistor en 1947.

1.3 La informática moderna

FORTRAN fue uno de los primeros lenguajes de programación. Fue desarrollado en 1957 por un equipo de ingenieros de IBM liderado por *John Backus*.



«FORTRAN» es un acrónimo de «Formula Translation».

El objetivo de FORTRAN era proporcionar un lenguaje de programación que permitiese a los científicos y matemáticos escribir programas de manera eficiente y fácilmente para ser utilizado en las computadoras de ese entonces.

PDP-1 fue un ordenador construido por la compañía Digital Equipment Corporation en 1959. El PDP-1 fue un ordenador de tiempo compartido, lo que significa que varios usuarios podían acceder al sistema al mismo tiempo y compartir los recursos del ordenador. Esto fue un gran avance en comparación con los ordenadores anteriores, que solían ser utilizados por un solo usuario a la vez.

«PDP-1» son las siglas de «Programmed Data Processor-1».

Figura 1.10: *Programmed Data Processor-1*

BASIC es un lenguaje de programación creado en el verano de 1964, con el objetivo de desarrollar un lenguaje de programación fácil de aprender y usar para estudiantes no especialistas y principiantes en la programación.

El sistema NLS fue un sistema desarrollado en 1968 por *Douglas Engelbart*. Fue una de las primeras demostraciones de una interfaz gráfica de usuario, con un puntero del ratón, sistema de hipertexto y ventanas.

«NLS» es un acrónimo de «oN-Line System».

También introdujo varias características que se consideran fundamentales en la computación moderna, como el procesamiento de textos, el correo electrónico, la videoconferencia y la colaboración en tiempo real.

Figura 1.11: NLS

W MULTICS fue un sistema operativo desarrollado en 1969 por un equipo liderado por el MIT y Bell Labs. El objetivo de MULTICS era crear un sistema operativo de tiempo compartido que pudiera ser utilizado por varios usuarios simultáneamente y ofrecer servicios avanzados, como el procesamiento de archivos, el manejo de bases de datos, manejo de permisos y el procesamiento de informes. A pesar de sus avances, MULTICS tuvo problemas financieros y técnicos que retrasaron su desarrollo y limitaron su adopción.

Tras haber participado en el desarrollo de MULTICS, *Ken Thompson* y *Dennis Ritchie* iniciaron en 1970 la creación de un nuevo sistema operativo. El proyecto fue bautizado originalmente como UNICS e inicialmente no tuvo apoyo económico por parte de los laboratorios Bell. Al año siguiente, *Dennis Ritchie* desarrolla el lenguaje de programación W *C*, un sistema diseñado para ser eficiente en términos de tiempo de ejecución y uso de recursos, y fácil de portar a diferentes plataformas de hardware.

Los sistemas operativos existentes hasta el momento eran propietarios y solo funcionaban en una plataforma específica, es por esto que en 1972, *Ken Thompson* y *Dennis Ritchie* decidieron reescribir el código de UNICS pero esta vez en lenguaje *C*, dando así origen a W *Unix*. Este cambio significaba que *Unix* podría ser fácilmente modificado para funcionar en otras computadoras y así otras variaciones podían ser desarrolladas por otros programadores.

En 1973 se desarrolló el ▶ *Xerox Alto*: un ordenador de tipo personal, con una interfaz gráfica de usuario, soporte para ventanas y un mouse. Además, tenía un sistema de gestión de archivos, un procesador de texto, una herramienta de dibujo y soporte para redes de computadoras. Aunque el W *Xerox Alto* nunca fue comercializado, muchas de sus características y conceptos se convirtieron en estándar en la industria de los ordenadores personales.

Figura 1.12: *Xerox Alto*

1.4 Primeras computadoras personales

La ▶ *Altair 8800* fue una de las primeras computadoras personales en ser comercializadas, lanzada en 1974. Fue un éxito de ventas y sentó las bases para el desarrollo de las computadoras personales que conocemos hoy en día. La W *Altair 8800* se vendía en kit y los usuarios debían armarla ellos mismos.

Figura 1.13: *Altair 8800*

En los primeros años del sistema *Unix* los Laboratorios Bell autorizaron a las universidades, a utilizar el código fuente y adaptarlo a sus necesidades. A partir de dicha iniciativa, en 1977 nace en la universidad de Berkley el sistema operativo W BSD.

«BSD» son las siglas de «Berkeley Software Distribution».

La W IBM PC fue una de las computadoras más importantes en la historia de la informática, ya que sentó las bases para el estándar de computadora personal que se utiliza en la actualidad. Fue introducida en 1981 y se convirtió rápidamente en el estándar de la industria para las computadoras personales. Una de las características más importantes de la IBM PC fue su arquitectura abierta. A diferencia de otras computadoras personales de la época, la IBM PC tenía un diseño abierto que permitía a los usuarios y terceros desarrollar sus propios productos y programas para ella. Esto ayudó a impulsar un gran ecosistema de desarrolladores y fabricantes de periféricos que crearon una gran variedad de software y hardware para la computadora.

«PC» son las siglas de «Personal Computer».

Figura 1.14: IBM PC

W *System V* es una versión del sistema operativo *Unix* desarrollado por Bell Labs en 1983. Fue una de las primeras versiones de *Unix* en ser comercializada y distribuida ampliamente, y tuvo un gran impacto en el desarrollo de los sistemas operativos tipo *Unix*. Aunque *Unix* se convirtió en un estándar en la industria de la computación, no se consideraba «*libre*» debido a las restricciones en su uso y distribución impuestas por AT&T.

Ese mismo año fue fundado el movimiento GNU por *Richard Stallman*, un programador y defensor de la libertad de software. El objetivo principal del movimiento W GNU es desarrollar un sistema operativo completo y gratuito basado en el estándar *Unix*, de manera que cualquier persona pueda usar, estudiar, compartir y modificar el software sin restricciones.

! Observación

El movimiento de software privativo había sido fundado con anterioridad en 1976 por *Bill Gates* a través de la denominada «W carta abierta a los aficionados».

W *X Windows* es un sistema de ventanas para sistemas tipo *Unix*. Fue desarrollado en el Massachusetts Institute of Technology en 1984. El objetivo principal de *X Windows* era proporcionar una interfaz gráfica que pudiera ser utilizado en una variedad de sistemas.

W *Minix* fue un sistema operativo educativo desarrollado en 1987 por *Andrew Tanenbaum*, para enseñar diseño y funcionamiento de sistemas operativos a estudiantes universitarios. Originalmente diseñado para ser utilizado en computadoras IBM PC y compatibles, *Minix* tenía un diseño similar al de *Unix*, pero con un conjunto reducido de herramientas y utilidades.

1.5 La era de la información

La década de 1990 marcó la comercialización de *Internet* y la expansión global. *Tim Berners-Lee* propuso la World Wide Web en 1989, y la adopción de navegadores web facilitó la navegación. Se establecieron empresas de tecnología, y se desarrollaron servicios como correo electrónico, motores de búsqueda y comercio en línea.

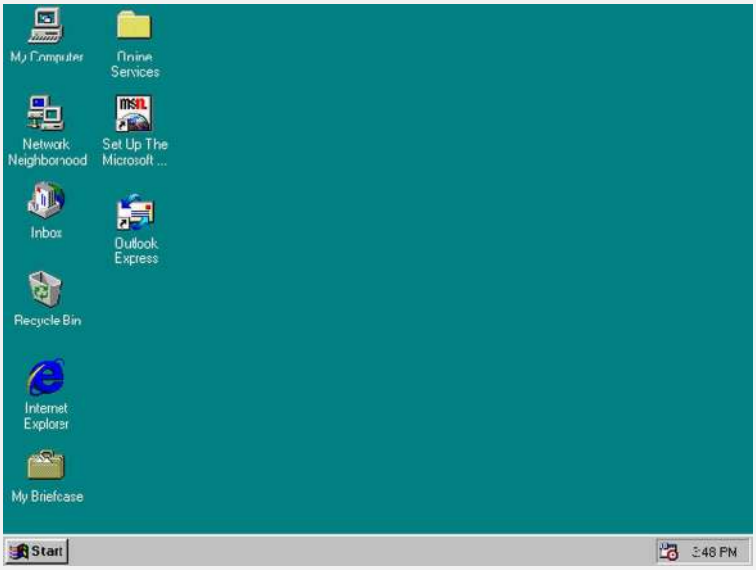
Mientras estudiaba informática en la Universidad de Helsinki en 1991, *Linus Torvalds* desarrolla un núcleo de sistema operativo (llamado W *Linux*) como proyecto personal, basándose en el diseño de *Minix*. Hasta el momento el proyecto GNU había desarrollado una amplia gama de software, incluyendo un compilador, un intérprete de línea de comandos y diversas herramientas de programación.

Sin embargo, faltaba un kernel, que es la parte del sistema operativo que administra los recursos del sistema, como la memoria y los procesos. Con el tiempo Linux se convirtió en una parte fundamental del proyecto GNU y en uno de los sistemas operativos de código abierto más utilizados y respetados de la industria tecnológica.

Python, creado por *Guido van Rossum* , fue concebido como un lenguaje de programación de alto nivel con un enfoque en la legibilidad del código y la facilidad de uso. Fue lanzado en febrero de 1991 y a lo largo de los años, ha ganado popularidad debido a su sintaxis clara y concisa, su versatilidad y su amplia gama de aplicaciones.

Windows 95, desarrollado por Microsoft y lanzado en agosto de 1995, fue un sistema operativo que marcó un hito significativo en la evolución de los sistemas operativos de Microsoft. Introdujo una interfaz gráfica de usuario más intuitiva, con elementos icónicos como el botón «*Inicio*», la barra de tareas, tecnología Plug and Play e Internet Explorer como navegador web.

Figura 1.15: W *Windows 95*



W *Deep Blue* fue una supercomputadora desarrollada por IBM que se hizo famosa por su capacidad para jugar al ajedrez. Fue especialmente conocida por su enfrentamiento contra el campeón mundial de ajedrez, *Garry Kasparov*, en 1996 y 1997. La victoria de *Deep Blue* marcó un hito en la historia de la inteligencia artificial y la computación, ya que fue la primera vez que una máquina derrotó a un campeón mundial de ajedrez.

Google es un motor de búsqueda en línea desarrollado por la empresa Alphabet Inc. Funciona como una herramienta que permite a los usuarios buscar información en la vasta cantidad de contenido disponible en la web. Fue fundada en 1998 por *Larry Page* y *Sergey Brin*, estudiantes de posgrado en la Universidad de Stanford.

Wikipedia es una enciclopedia en línea gratuita y colaborativa que permite a los usuarios crear, editar y actualizar contenido. Fue lanzada en 2001 y se basa en el principio, que permite la edición colaborativa de artículos por parte de usuarios de todo el mundo. Aunque la plataforma depende de la contribución voluntaria de sus usuarios, ha llegado a ser conocida por su vasta cantidad de información y su acceso abierto a la misma.

Ubuntu es una distribución de *Linux* basada en *Debian* y desarrollada por Canonical Ltd. Fue lanzada por primera vez en octubre de 2004 y desde entonces se ha convertido en una de las distribuciones de *Linux* más populares y ampliamente utilizadas en todo el mundo. W *Ubuntu* se enfoca en ofrecer una experiencia de usuario amigable y accesible, con una amplia gama de software preinstalado y una interfaz gráfica intuitiva.

Figura 1.16: Ubuntu 22.10

YouTube es una plataforma de compartición de videos en línea. Fundada en 2005, permite a los usuarios cargar, ver y compartir videos. Los usuarios pueden interactuar con los videos mediante comentarios, likes y suscripciones a canales. La plataforma se ha convertido en una fuente importante de contenido visual en línea y ha dado lugar a la creación de comunidades de creadores de contenido.

Android es un sistema operativo móvil que fue inicialmente creado en 2003 y adquirido por Google en 2005. W *Android* se ha convertido en el sistema operativo dominante para dispositivos móviles. Además de los teléfonos inteligentes, *Android* se utiliza en tabletas, televisores inteligentes, relojes inteligentes y otros dispositivos electrónicos.

GPT-3 es un modelo de lenguaje desarrollado por OpenAI a mitad de 2020. Es la tercera iteración de la serie GPT y representa uno de los modelos de lenguaje más grandes y avanzados hasta la fecha.

«GPT-3» son las siglas de «Generative Pre-trained Transformer 3».

Lo distintivo de GPT-3 es su capacidad para generar texto coherente y contextuado en respuesta a instrucciones dadas en lenguaje natural. Con 175 mil millones de parámetros, GPT-3 es notable por su enorme escala, lo que le permite abordar tareas diversas, como traducción de idiomas, redacción de textos, resolución de problemas matemáticos y más, sin una tarea específica de entrenamiento.

1.6 Ejercicios

Ejercicio 1.1. Investigue qué otros instrumentos se utilizaron para realizar cálculos antes de la *Pascalina*.

Ejercicio 1.2. Averigüe quién fue *Alan Turing* y cuál fue su principal contribución a las ciencias de la computación.

2

Arquitectura de la computadora

2.1 Gabinete

El W gabinete de la PC es una carcasa que cubre y protege los componentes de una computadora. Sus principales funciones son:

Protección Protege los componentes de la computadora de daños físicos, polvo y otros factores ambientales que pueden dañarlos.

Organización Los gabinetes de PC están diseñados para mantener todos los componentes en su lugar y organizados de manera eficiente. Esto facilita el trabajo del usuario y las tareas de mejora o reparación.

Refrigeración Ayudan a mantener los componentes frescos mediante la circulación de aire a través de la carcasa. Muchos gabinetes tienen ventiladores y otros sistemas de enfriamiento integrados para evitar el sobrecalentamiento de la computadora.

Figura 2.1: Gabinete Phobos Tg Xtech

2.2 Placa madre

La W placa madre es la pieza central de una computadora, encargada de conectar y comunicar todos los componentes esenciales del sistema. A través de sus conectores, la placa madre une la CPU, la memoria RAM, las unidades de almacenamiento, la tarjeta gráfica y otros dispositivos.

Además, distribuye la energía eléctrica necesaria a todos los componentes a través de los conectores de alimentación, y controla los puertos de entrada/salida que permiten la comunicación de la computadora con dispositivos externos, como los puertos USB, de audio y de red.

También incluye un chip de memoria ROM donde se almacena la BIOS, un programa que se encarga de configurar la computadora al encenderla y realizar pruebas iniciales del hardware.

La mayoría de las placas madre tienen un chip de audio integrado que proporciona capacidades de audio.

Figura 2.2: Placa madre

2.3 Fuente de alimentación

La W PSU o fuente de alimentación se encarga de convertir la corriente que recibe de la toma eléctrica en los diferentes voltajes necesarios para alimentar los componentes internos de la computadora.

«PSU» son las siglas de «Power Supply Unit».

Además protege los componentes de la computadora de sobretensiones, cortocircuitos y otros problemas eléctricos que pueden ocurrir. En caso de que se detecte una sobrecarga o falla, la PSU puede cortar el suministro de energía para proteger los componentes de la computadora.

Figura 2.3: Fuente de Alimentación

2.4 Microprocesador

La W CPU es el componente principal de una computadora. Es un chip integrado que se coloca en el zócalo de la placa madre y está compuesto por varios núcleos (o cores) que trabajan en conjunto.

«CPU» son las siglas de «Unidad Central de Procesamiento»

Es responsable de procesar y ejecutar los programas, manejar la entrada y salida de datos, y controlar los componentes del sistema, como la memoria RAM, el disco duro y las tarjetas de expansión. La velocidad y la capacidad de la CPU son factores clave que determinan el rendimiento general de una computadora.

Algunos procesadores modernos tienen gráficos integrados en su diseño. Estos gráficos integrados se denominan iGPU y están diseñados para proporcionar capacidades gráficas básicas para aplicaciones informáticas y de juegos de baja exigencia.

«iGPU» son las siglas de «Unidad de Procesamiento de Gráficos Integrados»

Figura 2.4: Intel Core i9

2.5 Memoria RAM

La memoria W RAM es un tipo de memoria que se utiliza para almacenar temporalmente los datos y programas que están en uso. Es un componente clave en el rendimiento de una computadora, ya que proporciona un acceso rápido a los datos y programas que el procesador necesita para operar.

«RAM» son las siglas de «Memoria de Acceso Aleatorio»

Cuando un programa se ejecuta, los datos y las instrucciones necesarios se cargan en la RAM desde el disco duro. La RAM permite que el procesador acceda rápidamente a estos datos y programas, lo que acelera el tiempo de ejecución y la velocidad de la computadora en general.

Es una memoria *volátil*, lo que significa que pierde todos los datos almacenados en ella cuando se apaga la computadora. Por lo tanto, es importante guardar los archivos y datos importantes en el disco duro o en otro dispositivo de almacenamiento persistente.

Figura 2.5: Memoria RAM Corsair Vengance DDR4

! Observación

Aunque el microprocesador también tiene una memoria volátil llamada *registros*, estos son mucho mas caros y, en consecuencia, pequeños.

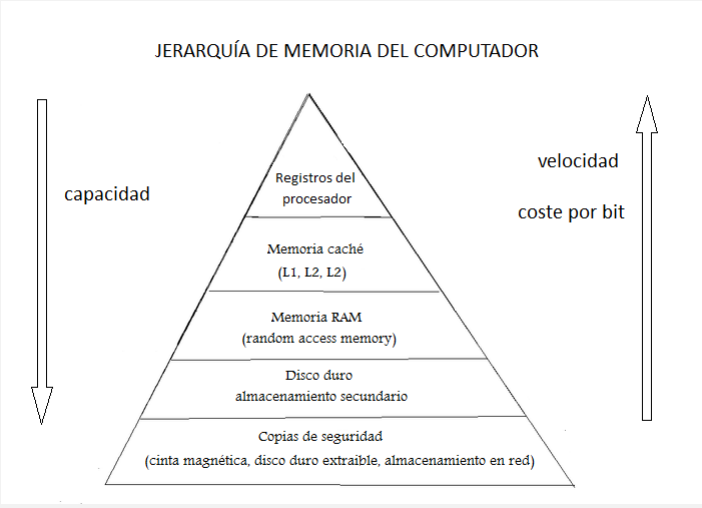
2.6 Memoria secundaria

Un disco duro es un dispositivo de almacenamiento de datos magnético que se utiliza para almacenar permanentemente archivos y programas. A diferencia de la memoria RAM, que es una memoria volátil, el disco duro los mantiene incluso después del apagado de la computadora.

Figura 2.6: Disco rígido Seagate Barracuda 1TB

Una de las principales diferencias entre el disco duro y la memoria RAM es la velocidad. Los discos duros son mucho más lentos en términos de velocidad de acceso, pues el brazo de lectura/escritura necesita moverse físicamente para acceder a los datos en los platos.

Figura 2.7: W Jerarquía de la memoria



Una W SSD es un dispositivo de almacenamiento de datos que utiliza memoria flash para almacenar permanentemente archivos y programas en la computadora.

«SSD» son las siglas de «Solid State Drive»

Figura 2.8: SSD Kingston

A diferencia de un disco duro tradicional, que utiliza platos magnéticos giratorios y cabezas de lectura/escritura para acceder a los datos, una SSD no tiene partes móviles. Como resultado, las SSD tienen un mejor rendimiento en términos de velocidad de lectura/escritura y tiempo de acceso.

Figura 2.9: NVMe Samsung

2.7 Placa de video

Una W placa de video, también conocida como tarjeta gráfica o GPU , es un componente de hardware que tiene como objetivo procesar y generar imágenes en la pantalla. Su función es liberar a la CPU de la tarea del procesamiento gráfico, permitiéndole concentrarse en otras tareas.

«GPU» son las siglas de «Graphics Processing Unit»

Figura 2.10: Tarjeta Gráfica NVIDIA RTX 2080 Ti

! Observación

Además de su uso en gráficos, en el campo de la I. A. se utilizan para entrenar y ejecutar redes neuronales debido a su arquitectura altamente paralela que les permite procesar grandes cantidades de datos de manera eficiente.

2.8 Teclado

El teclado es un dispositivo de entrada que permite a los usuarios introducir texto, números y comandos en una computadora. Es como una máquina de escribir moderna, pero con muchas más funciones.

Teclas especiales Las teclas especiales, a diferencia de las alfanuméricas, no introducen caracteres, sino que realizan funciones específicas o modifican el comportamiento de otras teclas. Alguna de ellas son:

- ⌨ Inicio: Generalmente, lleva el cursor al principio de la línea actual.
- ⌨ Fin: Sirve para llevar el cursor al final de la línea actual.
- ⌨ Re Pág: Desplaza la pantalla hacia arriba de a una página.
- ⌨ Av Pág: Análogo a ⌨ Re Pág.

Combinaciones especiales Además de las teclas especiales, también hay varias combinaciones de teclas que pueden ser útiles para navegar en un documento:

- ⌨ Ctrl+Inicio: te lleva al principio del documento.
- ⌨ Ctrl+Fin: te lleva al final del documento.
- ⌨ Ctrl+Izquierda: te mueve una palabra a la izquierda.
- ⌨ Ctrl+Derecha: te mueve una palabra a la derecha.
- ⌨ Ctrl+Arriba: te mueve al principio del párrafo anterior.
- ⌨ Ctrl+Abajo: te mueve al principio del párrafo siguiente.

Distribuciones Una distribución de teclado se refiere a la disposición de las teclas en un teclado, determinando qué carácter o función se asigna a cada tecla física.

Pueede pensarse como un mapa que define la relación entre las teclas que se pulsan y lo que aparece en la pantalla. Diferentes distribuciones se han diseñado para adaptarse a distintos idiomas, necesidades de escritura y preferencias de los usuarios.

Figura 2.11: Distribución de teclado estadounidense

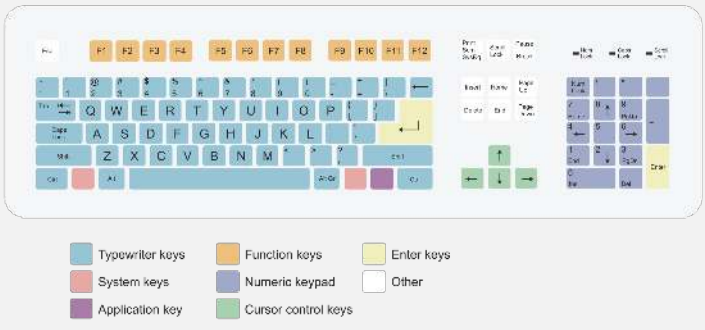


Figura 2.12: Distribución de teclado latinoamericana

2.9 Ejercicios

Ejercicio 2.1. Investigue y explique que es la «W Ley de Moore».

Ejercicio 2.2. Averigüe que son las distribuciones de teclado «W *Dvorak*» y «W *Colemak*».

3

Representación de la información

3.1 Unidades de información

Las unidades de información son medidas utilizadas en informática para cuantificar la cantidad de datos o información que se pueden almacenar o procesar. Estas unidades se basan en el sistema binario, ya que las computadoras trabajan con ceros y unos.

El bit (b) es la unidad más pequeña de información en un sistema digital. Representa un estado binario: 0 o 1. Un byte (B) está compuesto por 8 bits y es la únidad básica de información.

! Observación

Es importante prestar mucha atención a la diferencia de notación entre un bit (b minúscula) y un byte (B mayúscula).

Ejemplo 3.1. Para expresar un millón de bits (b) en bytes, basta con dividir por 8:

$$1{,}000{,}000b = \frac{1{,}000{,}000}{8}B = 125{,}000B$$

Ejemplo 3.2. De forma análoga podemos multiplicar por 8 para pasar de bytes a bits. Por ejemplo:

$$1{,}024B = 1{,}024 \cdot 8b = 8{,}192b$$

Ejemplo 3.3. Calculando potencias de dos, podemos conocer cuantos números diferentes pueden formarse con una cantidad de bits en particular. Por ejemplo, con 32 bits se pueden formar $2^{32} = 4{,}294{,}967{,}296$ números diferentes.

Además de las únidades báscias, también son de uso frecuente las unidades mayores, que representan cantidades más grandes de información. Cada una es una potencia de 2 (en el sistema binario) o de 10 (en el sistema decimal).

Cuadro 3.1: Unidades en sistema decimal

Nombre	Símbolo	Equivalencia	
Kilobyte	kB	10^3 Bytes	1,000 Bytes
Megabyte	MB	10^6 Bytes	1,000,000 Bytes (1,000 kB)
Gigabyte	GB	10^9 Bytes	1,000,000,000 Bytes (1,000 MB)
Terabyte	TB	10^{12} Bytes	1,000 GB
Petabyte	PB	10^{15} Bytes	1,000 TB

Cuadro 3.2: Unidades en sistema binario

Nombre	Símbolo	Equivalencia	
Kibibyte	KiB	2^{10} bytes	1,024 Bytes
Mebibyte	MiB	2^{20} bytes	1,048,576 Bytes (1,024 KiB)
Gibibyte	GiB	2^{30} bytes	1,073,741,824 Bytes (1,024 MiB)
Tebibyte	TiB	2^{40} bytes	1,024 GiB
Pebibyte	PiB	2^{50} bytes	1,024 TiB

! Observación

El sistema decimal se utiliza más frecuentemente en el almacenamiento comercial, ya que las empresas suelen usarlo para expresar capacidades más «redondas» y comprensibles.

Ejemplo 3.4. Resolviendo una simple ecuación, podemos pasar de GB a GiB:

$$500GB = 500 \cdot 10^9 B = xGiB = x2^{30} B$$
$$x = \frac{500 \cdot 10^9 B}{2^{30} B} \approx 465,66$$

Así, un disco duro anunciado como 500 GB (en sistema decimal) tendrá aproximadamente 465,66 GiB (en sistema binario).

3.2 Sistemas de numeración

Los sistemas de numeración son conjuntos de reglas y símbolos utilizados para representar cantidades o números. Cada sistema se basa en un conjunto específico de símbolos diferentes que se usan y se agrupan para formar números mayores. Cada sistema tiene sus propias aplicaciones y es útil en diferentes contextos, como la matemática, la computación y la ingeniería.

Sistemas no posicionales

Los sistemas de numeración no posicionales son aquellos en los que el valor de cada símbolo o dígito no depende de su posición dentro del número. En estos sistemas, cada símbolo tiene un valor fijo y constante, y los números se representan mediante la repetición o combinación de símbolos según ciertas reglas.

Ejemplo 3.5. El sistema de numeración unario es un sistema no posicional en el que los números se representan mediante la repetición de un único símbolo o marca. Cada símbolo representa una unidad, y el valor total del número es igual al número de símbolos presentes. Por ejemplo el número 3 se representaría como $|||$ y el 7 como $|||||||$.

Ejemplo 3.6. El sistema de numeración romano es un sistema no posicional utilizado en la antigua Roma para representar números. Se basa en un conjunto de símbolos con valores fijos que se combinan siguiendo reglas específicas para formar números. Por ejemplo el número 19 se representa como XIX y el 3999 como $MMMCMXCIX$.

Sistemas posicionales

Los sistemas de numeración posicionales son aquellos en los que el valor de un dígito depende tanto de su valor intrínseco como de la posición que ocupa dentro del número. En estos sistemas, se utiliza una base que determina cuántos símbolos se usan.

Fijada una base «b», un número se expresa como una combinación de dígitos (d_i) multiplicados por potencias de la base. La posición más a la derecha tiene el peso b^0, la siguiente b^1, luego b^2 y así sucesivamente.

La fórmula general para un número entero N de n dígitos es:

$$N = d_{n-1} \cdot b^{n-1} + d_{n-2} \cdot b^{n-2} + \cdots + d_1 \cdot b^1 + d_0 \cdot b^0 \qquad (3.1)$$

Para números con parte decimal, los pesos de las posiciones a la derecha del punto decimal son potencias negativas de la base, es decir:

$$N = d_{n-1} \cdot b^{n-1} + d_{n-2} \cdot b^{n-2} + \cdots + d_1 \cdot b^1 + d_0 \cdot b^0 + d_{-1} \cdot b^{-1} + d_{-2} \cdot b^{-2} + \cdots$$

! Observación

Cuando expresemos un número en una base que no sea 10, lo escribiremos entre paréntesis indicando la base como subindice. Así, el número $(101)_2$ no representa al número 101 en base 10, sino en base 2.

Ejemplo 3.7. El sistema decimal tiene base 10 (usa los dígitos 0 a 9). El número $345,67$ se descompone como: $3 \cdot 10^2 + 4 \cdot 10^1 + 5 \cdot 10^0 + 6 \cdot 10^{-1} + 7 \cdot 10^{-2}$.

Ejemplo 3.8. El sistema binario tiene como base 2 (usa los símbolos 0 y 1). El número $(1011,01)_2$ se descompone como:

$$(1011,01)_2 = 1 \cdot 2^3 + 0 \cdot 2^2 + 1 \cdot 2^1 + 1 \cdot 2^0 + 0 \cdot 2^{-1} + 1 \cdot 2^{-2} = \\ = 8 + 0 + 2 + 1 + 0 + 0,25 = 11,25$$

Ejemplo 3.9. La conversión de números entre el sistema octal (ocho símbolos) y el binario, es muy sencilla teniendo en cuenta la siguiente tabla:

Cuadro 3.3: Conversión octal/binario

Octal	Binario	Decimal	Octal	Binario	Decimal
$(0)_8$	$(000)_2$	0	$(4)_8$	$(100)_2$	4
$(1)_8$	$(001)_2$	1	$(5)_8$	$(101)_2$	5
$(2)_8$	$(010)_2$	2	$(6)_8$	$(110)_2$	6
$(3)_8$	$(011)_2$	3	$(7)_8$	$(111)_2$	7

Así pues, $(526)_8 = (101010110)_2$ y $(174)_8 = (001111100)_2$. Análogamente $(111000100)_2 = (704)_8$, $(11011000)_2 = (330)_8$, etc.

Ejemplo 3.10. El sistema hexadecimal tiene como base 16 (usa los números del 0 al 9 y las letras de la A a la F). El número $(1A)_{16}$ se descompone como:

$$(1A)_{16} = 1 \cdot 16^1 + A \cdot 16^0 = 1 \cdot 16^1 + A \cdot 16^0 = 16 + 10 = 26$$

! Observación

Puede convertir entre varios sistemas utilizando el siguiente «enlace».

3.3 Codificación de caracteres

Las W codificaciones de caracteres son sistemas que asignan un valor numérico (código) a cada carácter en un conjunto de caracteres, permitiendo que los textos sean representados, almacenados y transmitidos en forma digital.

Además, hay algunos símbolos en un sistema de codificación de caracteres que no tienen una representación visual directa en pantalla o en papel, pero que desempeñan funciones específicas dentro de un sistema. Estos caracteres se usan para controlar el formato, la estructura o el flujo de los datos, en lugar de ser representados como texto legible.

ASCII

El código W ASCII fue una de las primeras codificaciones utilizadas en computadoras. Representa letras, números y símbolos mediante valores numéricos de 7 bits (0-127).

«ASCII» son las siglas de «American Standard Code for Information Interchange»

! Observación

Es importante notar la diferencia entre el código de un caracter y el caracter que representa. Por ejemplo no hay que confundir el caracter nulo (codificado como 0) con el caracter «0» (codificado como 48).

Ejemplo 3.11. La palabra «*HOLA*» se codifica en ASCII de la siguiente manera:

ASCII Extendido

El ASCII estándar era suficiente para el inglés, pero se quedaba corto para otros idiomas que utilizan caracteres adicionales, como acentos, diéresis, o letras que no existen en el alfabeto inglés. Las variantes de ASCII extendido surgieron para cubrir estas necesidades, añadiendo caracteres específicos de diferentes regiones o idiomas.

Usa 8 bits (0-255), añadiendo 128 caracteres adicionales. Los primeros 128 caracteres (0-127) son idénticos al ASCII estándar, garantizando compatibilidad con sistemas antiguos.

Cuadro 3.4: Caracteres ASCII no imprimibles

Decimal	Hexadecimal	Caracter
0	00	Caracter nulo
1	01	Inicio de encabezado
2	02	Inicio de texto
3	03	Fin de texto
4	04	Fin de transmisión
5	05	Consulta
6	06	Acuse de recibo
7	07	Timbre
8	08	Retroceso
9	09	Tabulación
10	0A	Nueva linea
11	0B	Tabulación vertical
12	0C	Avance de página
13	0D	Retorno de carro
⋮	⋮	⋮

No existe un único «ASCII Extendido» estandarizado. En lugar de una única extensión universal, se crearon múltiples variantes de ASCII Extendido. Algunas de las variantes más comunes de ASCII Extendido son:

Latin-1 También llamada W ISO-8859-1, está diseñada para idiomas de Europa Occidental (inglés, francés, español, alemán, italiano, portugués, etc.). Incluye caracteres acentuados, símbolos comunes y algunos símbolos gráficos.

Windows-1252 También llamada W CP-1252, es utilizada, especialmente en sistemas Windows para Europa Occidental. Es muy similar a ISO-8859-1, pero con algunos caracteres adicionales.

! Observación

Es importante notar que si intentamos leer un mensaje con una codificación incorrecta, no obtendremos el resultado esperado.

Cuadro 3.5: Caracteres ASCII imprimibles

Decimal	Hexadecimal	Caracter
32	20	
33	21	!
34	22	"
35	23	#
⋮	⋮	⋮
48	30	0
49	31	1
50	32	2
⋮	⋮	⋮
65	41	A
66	43	B
⋮	⋮	⋮

Cuadro 3.6: Codificación en ASCII

H	O	L	A
72	79	76	65
1001000	1001111	1001100	1000001

Ejemplo 3.12. El texto «10 €» se codifica de la siguiente manera en CP-1252:

Cuadro 3.7: Codificación en CP-1252

1	0		**€**
49	48	32	**128**
00110001	00110000	00100000	**10000000**

Ejemplo 3.13. Si intentamos decodificar el texto anterior utilizando Latin-1, veremos algo similar a «10 ⊠», pues el caracter 128 en Latin-1 es un caracter no imprimible.

Ejemplo 3.14. Veamos como se codifica la palabra «*programación*»:

Cuadro 3.8: Codificación en Latin-1/Windows-1252

p	r	o	g	r	a	m	a	c	i	**ó**	n
112	114	111	103	114	97	109	97	99	105	**243**	110

UNICODE

A medida que la computación se globalizaba y la comunicación internacional se volvía más importante, se hizo evidente la necesidad de un estándar único y universal que pudiera representar todos los caracteres de todos los idiomas del mundo de manera consistente.

W Unicode surgió para solucionar el problema de las múltiples codificaciones de caracteres, creando un sistema unificado que permite a las computadoras manejar texto de cualquier idioma de manera consistente y confiable, allanando el camino para una comunicación digital verdaderamente global.

UTF

Unicode solo define el conjunto de caracteres. UTF es una familia de formatos de codificación que convierten los caracteres de Unicode en secuencias de bytes para su almacenamiento y transmisión en computadoras y redes. Los más comunes son:

«UTF» son las siglas de «Unicode Transformation Format»

UTF-32 Una codificación que utiliza siempre 4 bytes por carácter, incluso para caracteres ASCII que podrían representarse con 1 byte. Aunque simple, es menos eficiente en términos de almacenamiento.

UTF-8 Utiliza de 1 a 4 bytes para representar cada carácter Unicode. Es muy eficiente para caracteres comunes, pero también puede representar cualquier carácter Unicode. Es compatible con ASCII, lo que significa que los textos que solo usan caracteres ASCII se codifican de la misma manera en UTF-8 que en ASCII.

Ejemplo 3.15. Se pueden codificar expresiones matemáticas como «$0 \in \mathbb{N}$»:

Cuadro 3.9: Codificación en UTF-32

0				$\in$				$\mathbb{N}$			
0	0	0	48	0	0	34	8	0	0	33	21

Cuadro 3.10: Codificación en UTF-8

0	∈			ℕ		
48	226	136	136	226	132	149

Secuencias de escape

Una secuencia de escape es una combinación de caracteres que se usa para representar un símbolo o carácter de control que no se puede escribir directamente. También se utiliza para poder representar caracteres reservados por el sistema.

Estas secuencias comienzan con una barra invertida («\»), seguida de un carácter específico que indica la acción deseada.

Cuadro 3.11: Secuencias de escape mas frecuentes

Secuencia de escape	Caracter
\0	Caracter nulo
\a	Timbre
\b	Retroceso
\t	Tabulación
\n	Nueva linea
\r	Retorno de carro
\\	Barra invertida

3.4 Ejercicios

Ejercicio 3.16. ¿Cuál es el caudal en MB por segundo que maneja una linea de 80 Mbps?

Ejercicio 3.17. Convierta a decimal los siguientes números:

1. $(FF)_{16}$.
2. $(7F)_{16}$.

Ejercicio 3.18. Convierta a binario los siguientes números:

1. $(777)_8$.
2. $(644)_8$.

Ejercicio 3.19. Decodifique la siguiente palabra en ASCII: «67 65 78 84 79 32 70 69 76 73 90».

Ejercicio 3.20. ★ Imagine un sistema que almacena los nombres de archivos usando la códificación «W CP850», y un software que asume que el nombre está en Latin-1.

Los ejercicios marcados con ★ son de mayor dificultad.

1. ¿Cómo se llama un archivo codificado como «65 73 70 61 a4 6f 6c» dentro del sistema?
2. ¿Cómo interpreta el sowftare que se llama el archivo?
3. Suponga ahora que el software comprime el archivo y codifica el nombre en UTF-8. ¿Cómo se codificará?

4

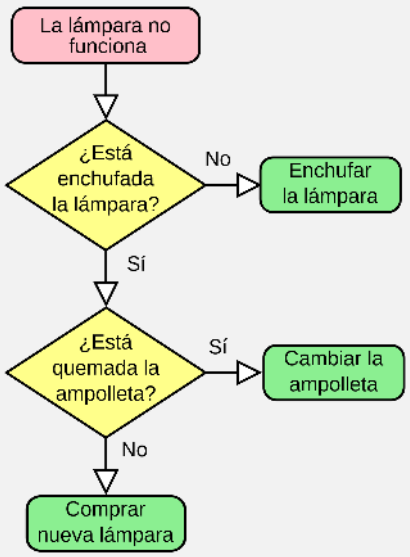
Conceptos de programación

4.1 Introducción

La programación es el proceso de diseñar, escribir, probar y mantener el código informático. En términos más simples, la programación es la forma en que los desarrolladores crean software, aplicaciones y sistemas.

Este proceso implica varios pasos, que incluyen la definición del problema a resolver, la identificación de los requisitos y especificaciones necesarias para crear la solución, la escritura del código en un lenguaje de programación, la realización de pruebas para asegurars el correcto funcionamiento del software y la documentación del código para que otros desarrolladores puedan entenderlo y trabajar en él.

Figura 4.1: Diagrama del flujo de un programa



4.2 Lenguaje de programación

Un lenguaje de programación es un conjunto de reglas, símbolos y palabras clave que se utilizan para comunicar instrucciones a una computadora. Estos lenguajes permiten a los programadores crear software, aplicaciones y sistemas, al proporcionar un medio para escribir código en un formato que la máquina pueda entender y ejecutar.

Los lenguajes de programación se utilizan para describir las acciones que una computadora debe realizar, como efectuar cálculos matemáticos, interactuar con periféricos, almacenar y recuperar datos, y responder a las entradas del usuario. Pueden variar en complejidad y enfoque, y algunos se centran en tareas específicas, mientras que otros son más generales y versátiles.

En esencia, un lenguaje de programación es una herramienta para crear software, permitiendo transformar ideas en código ejecutable.

! Observación

En esta materia aprenderemos el lenguaje de programación de «W `bash`». A lo largo de la carrera también aprenderán el lenguaje «W `python`».

4.3 Niveles de lenguajes

Los lenguajes de programación se pueden clasificar en tres niveles: alto, bajo y binario.

Alto Estos lenguajes son más cercanos al lenguaje natural humano y se utilizan para escribir programas complejos de manera más fácil y rápida, abstrayéndose de muchos detalles. Ejemplos de lenguajes de programación de alto nivel incluyen *Python*, *Java*, *C++*, *Ruby*, *PHP*, entre otros. Estos lenguajes tienen una sintaxis más amigable para el programador y se encargan de muchos de los detalles de bajo nivel, como la administración de memoria y el manejo de errores.

</> Código

```
print("Hola mundo!")
```

Bajo Estos lenguajes con cercanos al lenguaje de la máquina y se utilizan para escribir programas que interactúan directamente con el hardware de la computadora.

Ejemplos de lenguajes de programación de bajo nivel incluyen el lenguaje ensamblador y el lenguaje *C*. Estos lenguajes requieren que el programador tenga un conocimiento más detallado del hardware y de la forma en que se maneja la memoria.

</> Código

```
.global _start                  # gcc -nostdlib hola.s
.text

mensaje: .ascii  "Hola mundo!\n"
_start:
    mov $1, %rax                # La llamada a sistema 1 es write
    mov $1, %rdi                # El descriptor 1 es la salida estándar
    lea mensaje(%rip), %rsi     # Direccion de memoria del mensaje
    mov $12, %rdx               # Cantidad de bytes
    syscall                     # write(1, mensaje, 12)

    mov $60, %rax               # La llamada a sistema 60 es exit
    mov $0, %rdi                # Queremos devolver el numero 0
    syscall                     # exit(0)
```

BINARIO El lenguaje binario es el lenguaje de máquina utilizado por las computadoras para ejecutar programas. Este lenguaje está compuesto de ceros y unos, que representan instrucciones que el procesador puede entender y ejecutar directamente. Es muy difícil y tedioso para los programadores escribir directamente en lenguaje binario, por lo que se utilizan los lenguajes de programación de nivel alto y bajo para crear programas que luego se traducen a lenguaje binario.

</> Código

```
401000:    48 6f 6c 6c           # H o l a
401004:    20 6d 75 6e 64 6f     #   m u n d o
40100a:    21 0a                 # ! \n
40100c:    48 c7 c0 01 00 00 00  # mov $1, %rax
401013:    48 c7 c7 01 00 00 00  # mov $1, %rdi
40101a:    48 c7 c6 00 10 40 00  # mov $mensaje, %rsi
401021:    48 c7 c2 0c 00 00 00  # mov $12, %rdx
401028:    0f 05                 # syscall
40102a:    48 c7 c0 3c 00 00 00  # mov $60, %rax
401031:    48 c7 c7 00 00 00 00  # mov $0, %rdi
401038:    0f 05                 # syscall
```

4.4 Compiladores e intérpretes

Un intérprete y un compilador son dos tipos de programas que se utilizan para convertir el código fuente escrito por un programador en instrucciones ejecutables por una computadora.

Interprete Un intérprete es un programa que lee el código fuente de un programa y lo traduce en instrucciones ejecutables en tiempo real. El intérprete lee una línea de código fuente, la traduce a lenguaje de máquina y la ejecuta antes de pasar a la siguiente línea. Debido a que el intérprete realiza la traducción y la ejecución de cada línea a medida que se lee el código, puede ser más lento que un compilador. Sin embargo, el intérprete tiene la ventaja de que el programador puede ejecutar el código directamente sin necesidad de compilarlo previamente.

Compilador Un compilador, por otro lado, es un programa que traduce todo el código fuente a lenguaje de máquina en un solo paso, generando un archivo ejecutable que puede ser utilizado por la computadora sin necesidad de leer el código original nuevamente. El compilador lee todo el código fuente del programa y lo traduce en un archivo ejecutable que puede ser utilizado por la computadora sin necesidad de leer el código fuente original nuevamente. Debido a que el código fuente se traduce y compila solo una vez, el código compilado se ejecuta generalmente más rápido que el código interpretado. Sin embargo, el proceso de compilación puede llevar más tiempo que el proceso de interpretación.

Figura 4.2: Tiempo de traducción y ejecución

! Observación

En los lenguajes compilados, los tiempos de traducción son computados por el desarrollador. En cambio en los lenguajes interpretados, son computados por el usuario final.

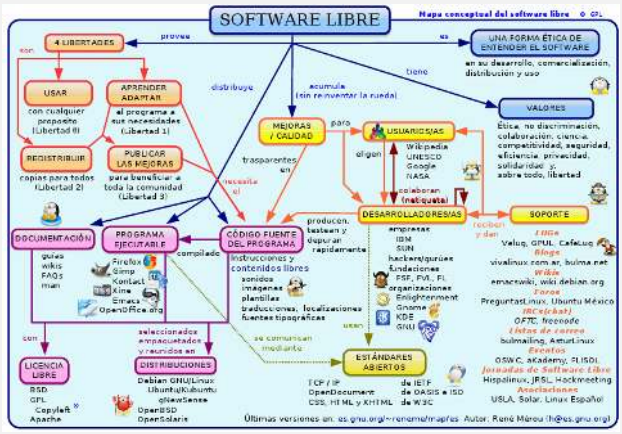
4.5 Otros conceptos

Código fuente El código fuente de un programa es el conjunto de instrucciones escritas por un programador en un lenguaje de programación determinado; es el «*texto*» legible por humanos que se escribe en un archivo de código fuente y que debe ser traducido o compilado en lenguaje de máquina para que la computadora pueda entenderlo y ejecutar el programa.

Código abierto El término «*código abierto*» se refiere a un tipo de software en el cual el código fuente está disponible públicamente y puede ser modificado por cualquier persona. En el modelo de código abierto, los desarrolladores de software pueden trabajar para mejorar y expandir el software existente.

Software libre El software libre es aquel que se distribuye con una W licencia que permite a los usuarios tener acceso al código fuente del programa y redistribuir versiones modificadas del mismo sin tener que pagar regalías o cargos adicionales. En general, dicha licencia establece los términos y condiciones bajo los cuales se puede usar, modificar y distribuir el software.

Figura 4.3: Mapa conceptual del software libre



! Observación

El software libre suele estar disponible gratuitamente, pero no es obligatorio que sea así; por lo tanto no hay que asociar software «*libre*» a «*gratuito*» ya que, conservando su carácter de libre, puede ser distribuido comercialmente.

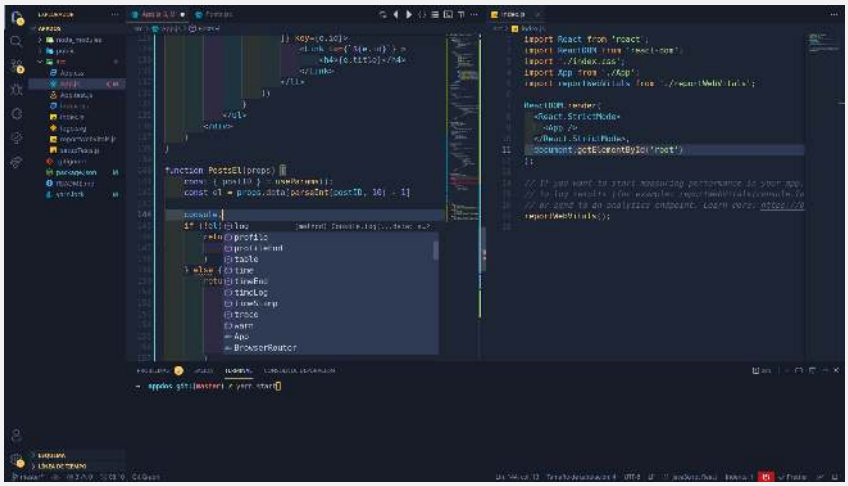
IDE Un IDE es un conjunto de herramientas y características que se combinan en una única aplicación para hacer más fácil y eficiente el proceso de desarrollo de software.

«IDE» son las siglas de «Integrated Development Environment»

Típicamente incluye un editor de código que proporciona herramientas de resaltado de sintaxis y autocompletado, para facilitar la escritura de código. Además, un IDE suele incluir un depurador que ayuda a identificar y corregir errores en el código.

También puede tener herramientas de compilación y construcción, que permiten compilar y empaquetar el código en un archivo ejecutable o en otro formato adecuado para su distribución.

Figura 4.4: *Visual Studio Code*



4.6 Ejercicios

Ejercicio 4.1. Investiga y explica las «W *cuatro libertades*» del software libre definidas por la Free Software Foundation.

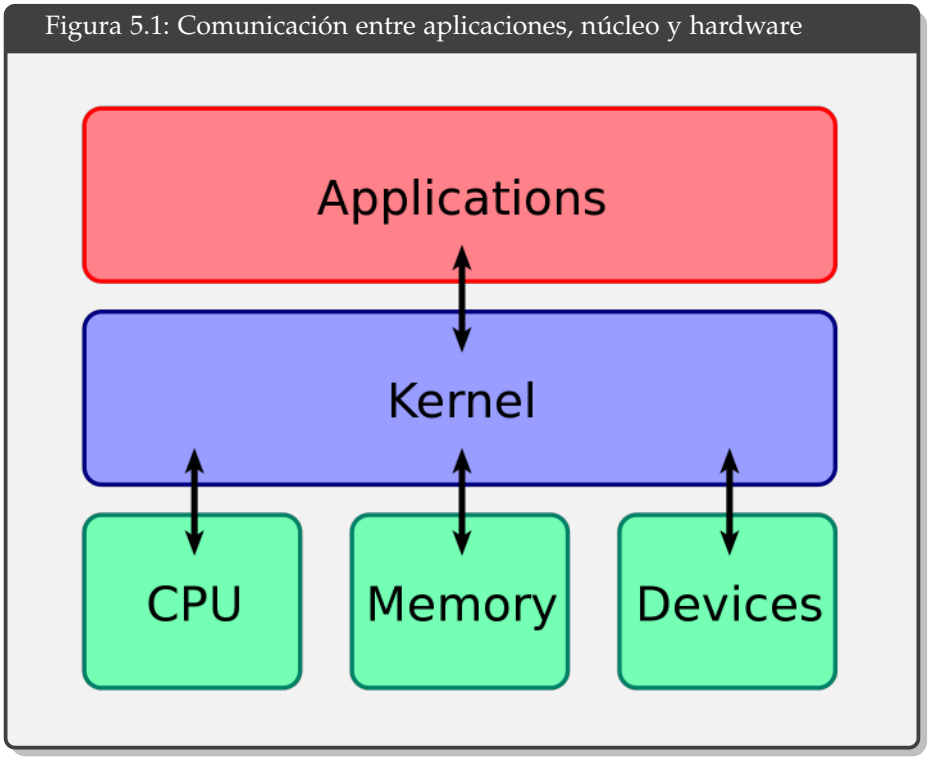
5

Sistemas Operativos

5.1 Descripción

Un sistema operativo es un conjunto de programas y herramientas que controlan y coordinan las actividades de una computadora o dispositivo electrónico, y permiten a los usuarios interactuar con el hardware y el software de manera sencilla y eficiente. Está compuesto por un núcleo (o *kernel*) que tiene control total sobre el hardware en el que corre, y una serie de programas utilitarios que se comunican con el.

Figura 5.1: Comunicación entre aplicaciones, núcleo y hardware



Las funcionalidades principales de un sistema operativo moderno incluyen:

Abstracción de hardware El sistema operativo oculta los detalles específicos del hardware de la computadora a las aplicaciones y los usuarios. Esto permite permite a las aplicaciones interactuar con el hardware de manera consistente y simplificada, sin tener que preocuparse por los detalles internos del hardware subyacente.

Gestión del hardware El sistema operativo es responsable de gestionar el hardware, como el procesador, la memoria, el disco duro, la tarjeta gráfica, entre otros. Controla cómo se utilizan estos recursos y asigna la cantidad adecuada de memoria y procesador a cada aplicación.

Interfaz de usuario El sistema operativo proporciona una interfaz de usuario que permite a los usuarios interactuar con la computadora y ejecutar aplicaciones.

Gestión de archivos El sistema operativo se encarga de gestionar los archivos y carpetas, permitiendo a los usuarios crearlos, modificarlos, copiarlos y eliminarlos.

Multitarea Un sistema operativo permite que varias aplicaciones se ejecuten al mismo tiempo y asigna los recursos necesarios para que funcionen correctamente, dando la ilusión de paralelismo.

Multiusuario Un sistema operativo puede ser utilizado por varios usuarios al mismo tiempo y garantiza que cada usuario tenga sus propios archivos y configuraciones.

5.2 Núcleo

El *kernel* (o núcleo) de un sistema operativo es la parte central y más fundamental del mismo. Es responsable de controlar el acceso a los recursos del hardware, gestionar los procesos, la memoria y la entrada/salida. Además proporciona una interfaz para que las aplicaciones interactúen con el hardware del sistema.

El núcleo se ejecuta en modo privilegiado, lo que significa que tiene acceso directo al hardware y puede ejecutar instrucciones que otros programas no pueden.

Para casi cualquier tarea las aplicaciones de usuario necesitan pedirle permiso al kernel, mediante una instrucción denominada «W *llamada a sistema*». Cuando ocurre una llamada a sistema, el CPU detiene la ejecución del programa y comienza a ejecutar la funcionalidad del núcleo requerida, luego de la cual se reanuda el flujo original del programa.

5.3 Proceso de arranque

Durante el arranque de una PC, ocurren varios pasos importantes que permiten que el sistema operativo se inicie correctamente y la computadora esté lista para su uso.

Al presionarse el botón de arranque se activa la fuente de alimentación de la computadora, la cual suministra la energía necesaria para que la placa madre comience a funcionar. A partir de aquí, comienza un proceso que consiste en varias etapas:

POST (Power On Self Test) La placa madre realiza un autodiagnóstico para verificar que todos los componentes de hardware de la computadora estén funcionando correctamente. Si detecta algún problema, emitirá un mensaje de error y detendrá el proceso de arranque.

Figura 5.2: Etapa de Autodiagnóstico

Boot Loader A continuación la placa madre debe cargar un programa llamado «*cargador de arranque*», el cual tiene como objetivo principal cargar el núcleo del sistema operativo. En los sistemas Linux el mas utilizado es «*GRUB*»; y «`bootmgr`» es el proporcionado por los sistemas modernos de *Windows*.

Figura 5.3: *GRUB*

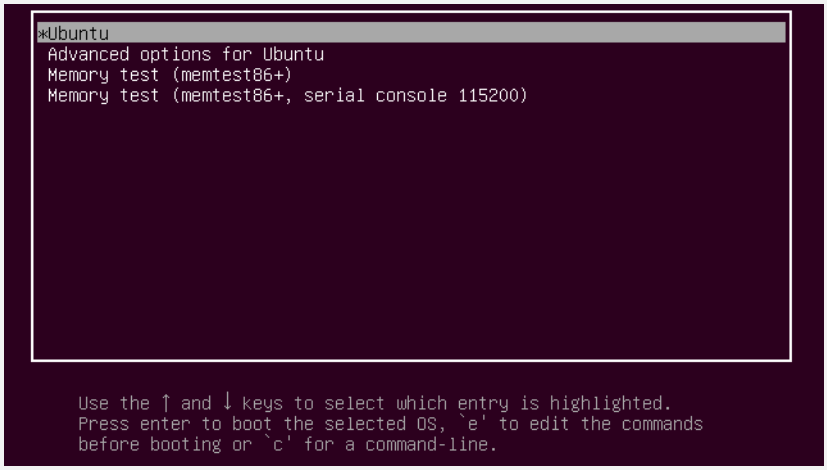
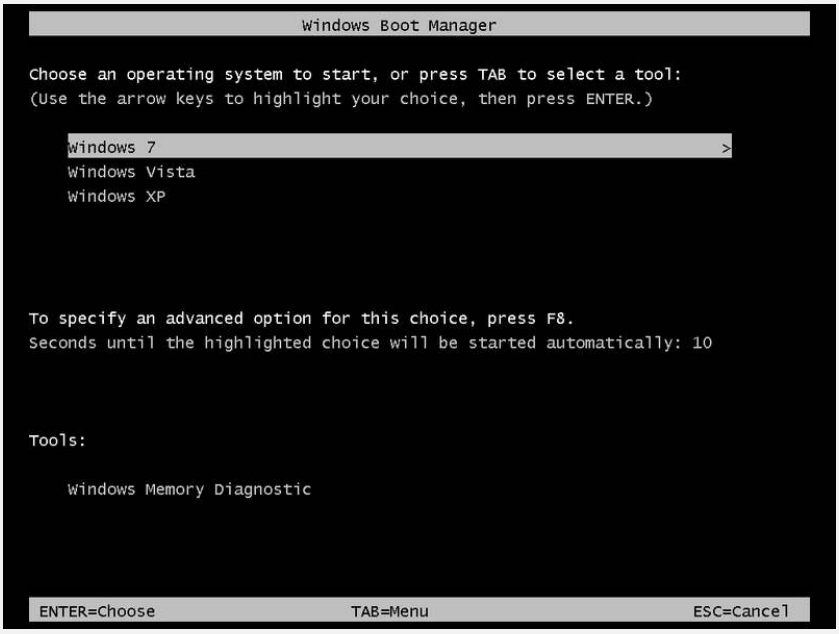


Figura 5.4: bootmgr



Núcleo A continuación, el núcleo del sistema operativo toma el control de la computadora. Durante sus tareas iniciales se encargará de identificar el hardware disponible, cargar los controladores necesarios y montar el sistema de archivos del sistema. Finalmente dará comienzo al primer programa de usuario, a partir del cual se ejecutaran todos los demás programas.

Init En los sistemas *Linux*, el programa inicial del sistema operativo se llama «`init`». Init se encargará de cargar los scripts de arranque del sistema, así como también ejecutar los servicios esenciales para el funcionamiento del mismo, y proveer al usuario de un entorno gráfico o de linea de comandos.

! Observación

En la actualidad se desarrolló «W *systemd*» para reemplazar el sistema de inicio (`init`) heredado de los sistemas operativos estilo UNIX System V y Berkeley Software Distribution (BSD).

5.4 Interfaz de usuario

En términos generales, la interfaz de usuario (UI) es el punto de interacción y comunicación entre un ser humano y un programa. Una buena interfaz de usuario facilita la interacción de forma intuitiva, eficiente y satisfactoria, permitiendo al usuario realizar sus tareas y alcanzar sus objetivos de manera efectiva.

Existen principalmente tres tipos de interfaz:

GUI Una interfaz gráfica de usuario «GUI» , es una forma de interactuar con un programa o sistema operativo mediante el uso de elementos gráficos, como ventanas, iconos, botones y menús.

«GUI» son las siglas de «Graphical User Interface»

Están diseñadas para ser fáciles de aprender y usar, incluso para usuarios sin experiencia en informática. Esto permite a los usuarios realizar acciones de forma intuitiva, haciendo clic, arrastrando y soltando objetos en la pantalla.

! Observación

Es importante notar que las interfaces gráficas en *Linux* son un programa de usuario opcional.

Figura 5.5: *KDE Plasma*

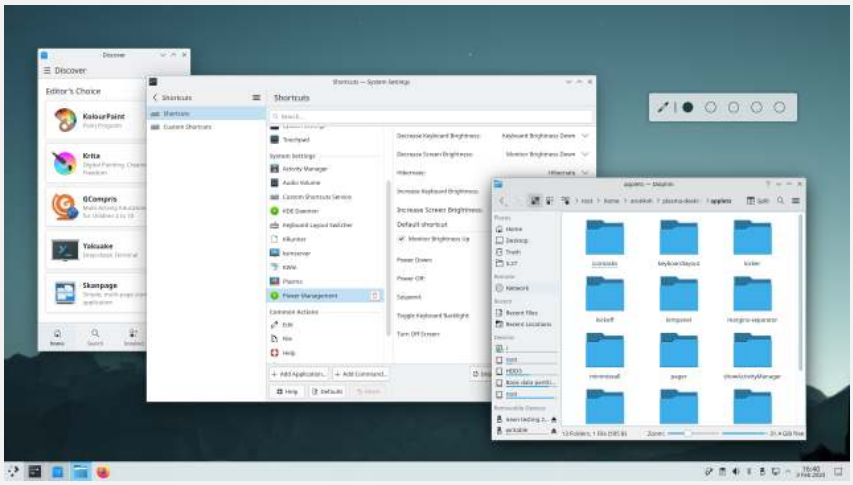
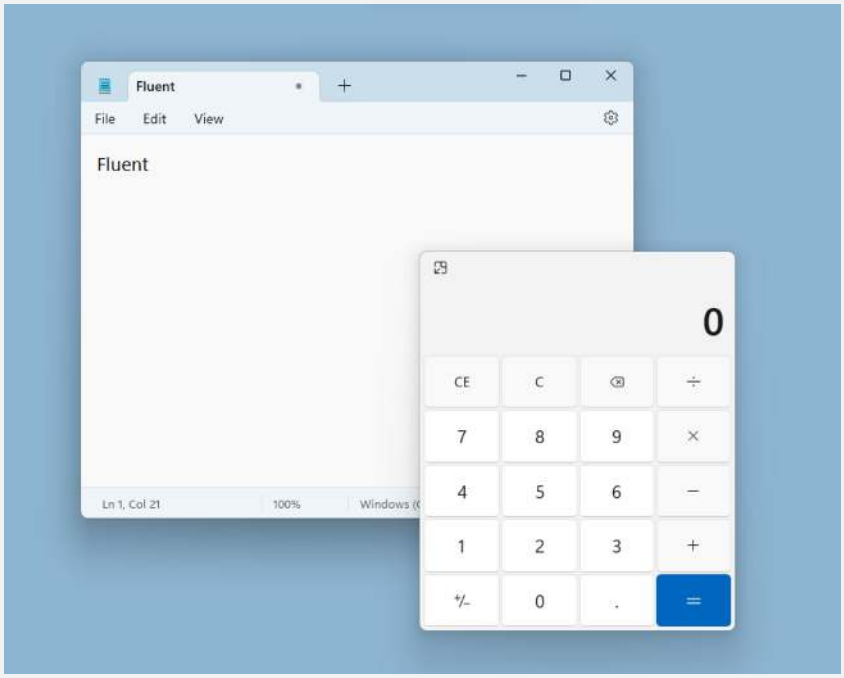


Figura 5.6: *Fluent Design*



CLI Una «CLI» es una interfaz basada en texto, donde el usuario interactúa con el sistema operativo escribiendo comandos específicos en una línea de comandos.

«CLI» son las siglas de «Interfaz de Línea de Comandos»

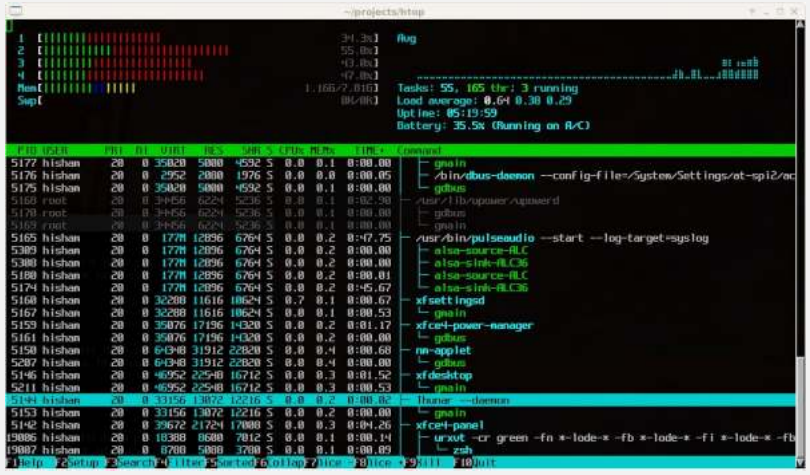
Requieren un mayor conocimiento técnico y la memorización de comandos, pero consumen menos recursos del sistema en comparación con las GUIs.

> **! Observación**
>
> En esta materia estudiaremos principalmente una CLI.

TUI Una TUI es una interfaz híbrida que combina elementos de las CLIs y las GUIs. Se ejecuta en un entorno de texto, pero utiliza caracteres y colores para crear una apariencia más visual y estructurada, con elementos como menús, cuadros de diálogo y ventanas de texto.

«TUI» son las siglas de «Text-based User Interface»

Figura 5.7: htop



5.5 Terminal

A menudo se utilizan términos como «*terminal*», «*consola virtual*», «*emulador de terminal*» o «*intérprete de línea de comandos*» de forma indistinta, lo que puede llevar a cierta confusión. A continuación, se explican las diferencias entre estos términos:

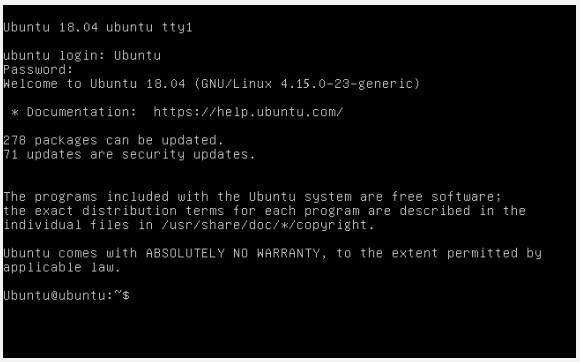
Terminal Se refiere a el o los dispositivos físicos que se utilizan para interactuar con un ordenador mediante la entrada y salida de texto. En la actualidad está compuesta principalmente por el teclado y el monitor.

Consola virtual Es una aplicación implementada dentro del núcleo que provee acceso al sistema simulando una terminal de teletipo. En los sistemas tipo *Unix* **con entorno gráfico** se puede acceder a ellas presionando ⌨ Ctrl+Alt+F1, ⌨ Ctrl+Alt+F2, etc. En los sistemas **sin entorno gráfico** la combinación de teclas es ⌨ Alt+F1, ⌨ Alt+F2, etc.

! Observación

Aunque el entorno gráfico ha ganado popularidad frente a las interfaces de texto, las consolas virtuales siguen siendo utilizadas por varias razones como el uso eficiente de recursos, acceso remoto, tareas de mantenimiento y recuperación.

Figura 5.8: Consola virtual en *Ubuntu*

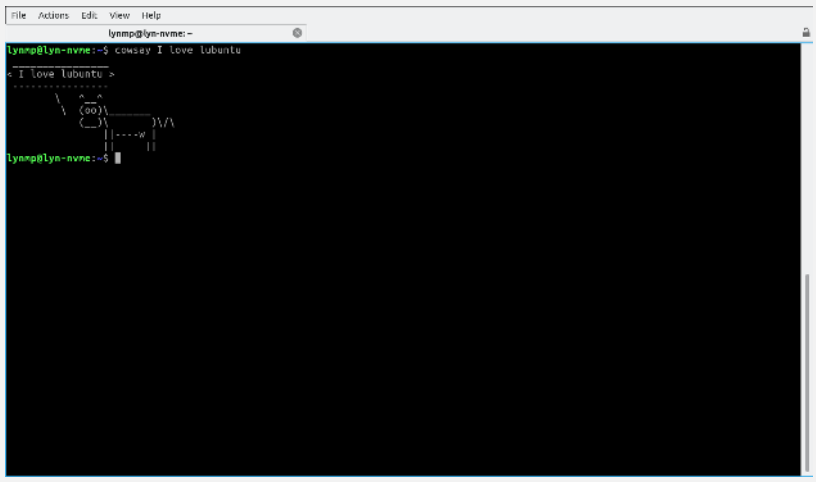


Emulador de terminal Es un programa de usuario que permite interactuar con un sistema operativo a través de una ventana en un entorno gráfico. Los emuladores de terminal son comúnmente utilizados para acceder a sistemas remotos o para ejecutar aplicaciones de línea de comandos en sistemas operativos.

! Observación

En un emulador de terminal las combinaciones de teclas habituales para copiar y pegar no funcionan. Debemos usar en su lugar ⌨ Ctrl+Shift+C (para copiar) y ⌨ Ctrl+Shift+V (para pegar).

Figura 5.9: `Qterminal` en *Lubuntu*



Shell También llamado «*interprete de linea de comandos*», es un programa que permite a un usuario interactuar con el sistema operativo mediante la ejecución de comandos a través de una interfaz de línea de comandos.

> **! Observación**
>
> Alguno de los interpretes de linea de comandos mas populares en los sistemas Linux son: «W `sh`», «🔗 `bash`», «🔗 `zsh`» y «🔗 `fish`». En los sistemas operativos de Miscrosoft se destacan: «W `COMMAND`», «W `cmd`» y «🔗 `PowerShell`».

5.6 Procesos

Un proceso es una instancia de un programa en ejecución. Durante su ciclo de vida, atraviesa diferentes etapas las cuales incluyen:

Creación En esta etapa, el sistema operativo crea un nuevo proceso cuando se inicia un programa o se solicita mediante una llamada al sistema. Durante la creación, se asignan recursos como espacio de memoria, identificadores de proceso y otros atributos necesarios para la ejecución.

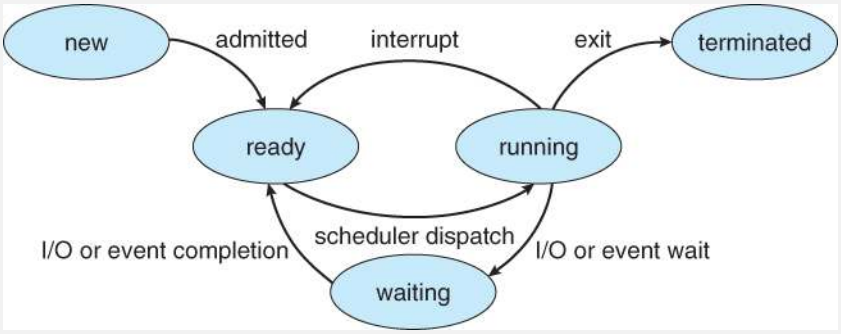
Listo En este estado, el proceso está cargado en memoria a la espera de ser seleccionado por el planificador del sistema operativo para su ejecución.

Ejecución Cuando un proceso es seleccionado por el planificador del sistema, se mueve al estado de «*ejecución*». En esta etapa, el procesador ejecuta las instrucciones del programa y realiza las tareas definidas por el proceso. El proceso continúa en este estado hasta que le toque ejecutar a otro programa o se bloquee esperando alguna operación de entrada/salida.

Bloqueado Un proceso puede entrar en el estado «*bloqueado*» si necesita esperar por alguna operación de entrada/salida, como leer datos de un archivo o esperar una respuesta de red. Cuando esto ocurre, el proceso se suspende temporalmente y se libera el uso del procesador.

Finalización Cuando un proceso ha terminado de ejecutar todas sus instrucciones, o ha sido terminado de manera forzada por el sistema operativo, entra en el estado de «*finalización*», liberando los recursos asignados realizando la limpieza necesaria.

Figura 5.10: Etapas de un proceso



Cabe destacar que el sistema operativo puede realizar transiciones entre los diferentes estados del proceso según las decisiones del planificador y los eventos que ocurren durante la ejecución.

5.7 Distribuciones

Las distribuciones de Linux son sistemas operativos basados en el kernel de *Linux*, que están compuestos por una combinación de software libre y de código abierto, como aplicaciones, controladores, herramientas de gestión de paquetes, etc.

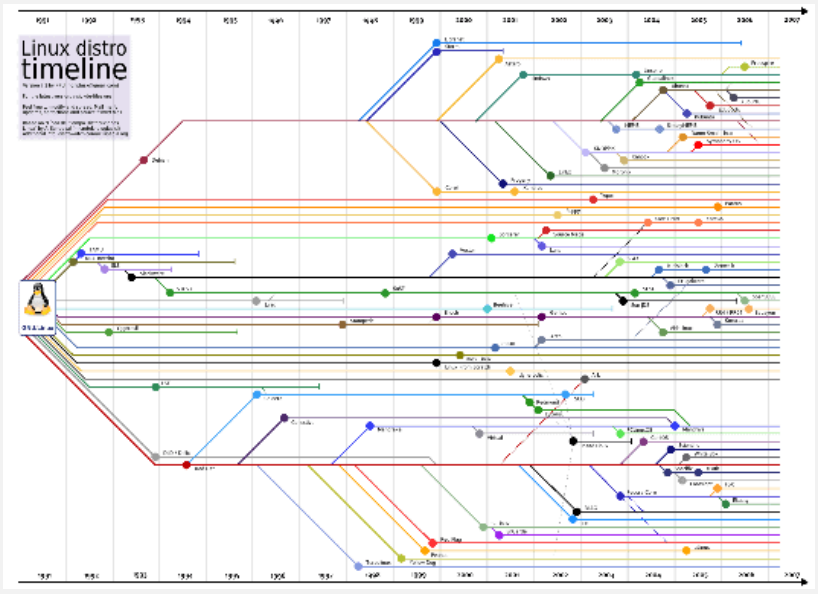
Existen muchas distribuciones de *Linux* diferentes, como *Debian*, *Ubuntu*, *Red Hat*, *Arch Linux*, entre otras. Cada distribución tiene sus propias características, objetivos y filosofía, y están diseñadas para satisfacer las necesidades de diferentes usuarios y aplicaciones.

! Observación

El concepto de distribución de *Linux* se hereda de los sistemas operativos estilo *Unix* donde las universidades comenzaron a desarrollar sus propias versiones, como *System V* y BSD.

Las distribuciones de *Linux* existen porque el software de código abierto permite a los usuarios y desarrolladores acceder, modificar y distribuir el código fuente del software. Esto ha permitido que muchas personas y comunidades puedan desarrollar y distribuir sus propias versiones personalizadas de *Linux*. Además, al ser un sistema operativo altamente personalizable y adaptable, cada distribución puede estar diseñada para satisfacer las necesidades específicas de diferentes usuarios, como por ejemplo para usuarios de servidores, programadores, usuarios de escritorio, entre otros.

Figura 5.11: Linea de tiempo de distribuciones de *Linux*



5.8 Proceso de apagado

Antes de finalizar la ejecución del sistema operativo, se inicia un proceso que cierra todos los programas y servicios que se están ejecutando. Luego, se guardan todos los datos pendientes y se asegura que todos los dispositivos de almacenamiento estén en un estado seguro antes de apagar la alimentación. Finalmente, se envía una señal al hardware para que se apague por completo y se desconecte la alimentación en caso de ser necesario.

Hay varias opciones disponibles para «*apagar*» una computadora, cada una con diferentes efectos.

Apagado Cuando se selecciona la opción de «*apagar*», la computadora cierra todos los programas y procesos en ejecución y se apaga completamente. La próxima vez que se encienda la computadora, se iniciará el proceso de arranque completo.

Reinicio Al seleccionar la opción «*reiniciar*», la computadora cierra todos los programas y procesos en ejecución, se apaga brevemente y luego se reinicia automáticamente. Esta opción es útil para solucionar problemas de hardware o software y para actualizar el sistema operativo.

Hibernación La opción de «*hibernar*» guarda todos los datos y configuraciones del sistema en el disco duro y luego apaga la computadora. Cuando se vuelve a encender la computadora, el sistema restaura automáticamente los datos y la configuración de la sesión anterior.

> **! Observación**
>
> La información almacenada en la memoria RAM se guarda en un archivo o en una partición exclusiva dedicada a tal propósito.

Suspensión La opción «*suspender*» pone la computadora en un estado de bajo consumo de energía, dejando alimentada solamente la memoria RAM. De esta manera los programas y procesos en ejecución se conservan y la computadora puede volver a su estado anterior cuando se reanude la actividad. Esta opción es útil para ahorrar energía y reanudar rápidamente el trabajo en curso.

> **! Observación**
>
> Si se llega a producir un corte en el suministro eléctrico, el estado de la computadora se pierde pues la RAM será incapaz de retener su información.

5.9 Ejercicios

Ejercicio 5.1. ✎ ¿Por qué no existen diferentes distribuciones de *Windows*?

Los ejercicios marcados con ✎ han sido ejercicios de examen.

6

Sistema de archivos

Aunque la «W *memoria principal*» es esencial para el funcionamiento de una computadora, esta es de acceso volátil, lo que significa que se borra cuando se apaga la computadora. Por lo tanto, la memoria RAM solo puede almacenar temporalmente los programas y datos que están en uso en ese momento.

En este capítulo explicaremos como funcionan las llamadas «W *memorias secundarias*» de la computadora, como el disco rígido o una unidad de estado solido.

6.1 Acceso aleatorio y secuencial

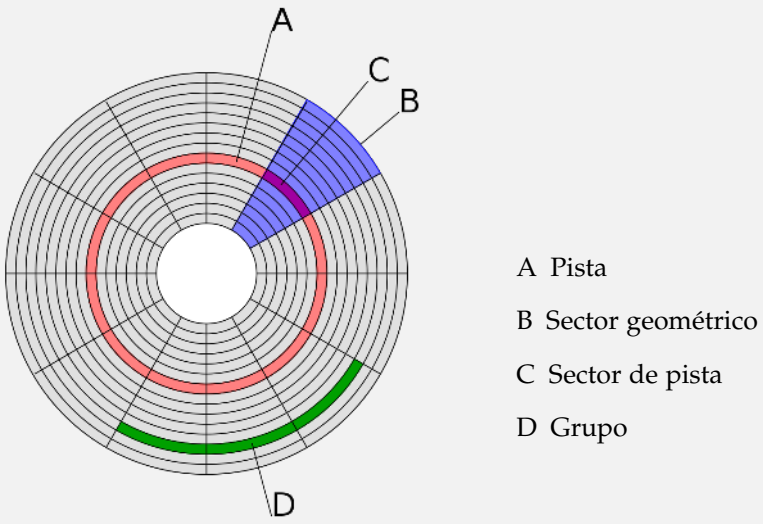
Un dispositivo de acceso aleatorio, como una unidad de estado sólido o USB, permite el acceso directo a cualquier ubicación en el medio de almacenamiento. Esto significa que se puede acceder a cualquier archivo o dato almacenado en el dispositivo sin tener que pasar por los datos que están almacenados «*antes*» o «*después*» en el dispositivo.

! Observación

Puesto que los dispositivos de acceso aleatorio son muy rápidos, son ideales para almacenar programas y mejorar el rendimiento general de la computadora.

Por otro lado, un dispositivo de acceso secuencial, como un disco rígido o DVD, requiere que los datos se lean en secuencia, desde el principio del medio de almacenamiento hasta el final. Esto significa que para acceder a un archivo o dato específico, el dispositivo debe leer los datos almacenados antes de llegar al archivo o dato deseado.

Figura 6.1: Estructura de un disco



! Observación

Las arquitecturas de computadora modernas suelen utilizar además de una memoria de acceso directo, otra de acceso secuencial. Esto es puesto que pese a son mas lentas, son mas baratas y de mayor capacidad, lo que los hace ideales como medios de respaldo.

6.2 Descripción

Un sistema de archivos es una estructura lógica y organizada de datos que se utiliza para almacenar y recuperar información en un medio de almacenamiento. Sin un sistema de archivos, los datos colocados en un medio de almacenamiento serían un gran cuerpo de datos sin manera de saber dónde termina un dato y comienza el siguiente.

Sus principales funciones son:

Gestión del espacio El sistema de archivos se encarga de gestionar la manera en que los archivos son almacenados en un dispositivo de almacenamiento de datos. Esto incluye entre otras cosas, la asignación de espacio en disco para cada archivo, la administración del espacio libre y la organización de los datos de manera eficiente.

Organización El sistema de archivos organiza los archivos en una jerarquía de directorios y subdirectorios, lo que facilita su búsqueda y recuperación.

Permisos El sistema de archivos puede incluir medidas de seguridad para proteger los datos almacenados en el medio de almacenamiento, como permisos de acceso, y encriptación de datos.

Metadatos Un sistema de archivos almacena una variedad de metadatos que proporcionan información sobre los archivos y directorios que se almacenan en el dispositivo de almacenamiento. Algunos de los metadatos más comunes que se almacenan en un sistema de archivos son el nombre del archivo, su tamaño, fecha de creación y modificación, etc.

Integridad Un sistema de «W *journaling*» (o diario) es una técnica utilizada para proteger la integridad de los datos. Este sistema funciona registrando todos los cambios que se realizan en el sistema de archivos en un registro o diario antes de que los cambios sean efectuados en el sistema de archivos. En caso de un fallo del sistema o un corte de energía, el sistema de journaling puede utilizar la información en el registro para volver al estado anterior del sistema de archivos, evitando así la pérdida de datos o la corrupción del sistema de archivos.

! Observación

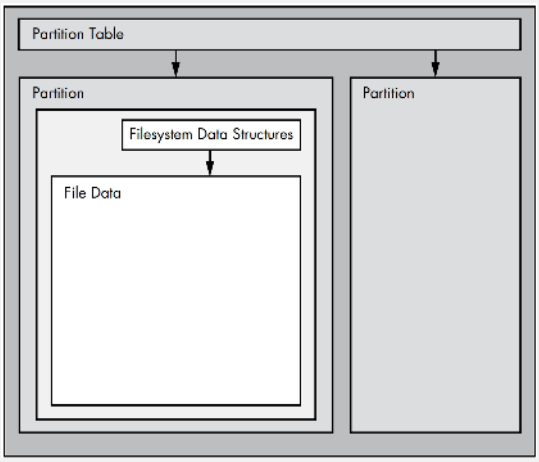
En los sistemas *Linux*, el sistema de archivos mas utilizados es «W *ext4*», mientras que en los sistemas *Windows* se utiliza «W NTFS».

6.3 Particiones

Una partición es una sección lógica de un disco duro u otro dispositivo de almacenamiento que se trata como si fuera un disco separado. Una partición se crea mediante la división del espacio de almacenamiento disponible en el dispositivo en secciones separadas. Cada partición debe tener su propio sistema de archivos y puede contener archivos y datos independientes de las demás particiones.

Las particiones se utilizan para varios fines, como separar el sistema operativo y los archivos del usuario, o para crear diferentes áreas de almacenamiento para diferentes propósitos, como la música, los documentos y las imágenes. También se pueden utilizar para instalar varios sistemas operativos en una sola unidad de disco duro.

Figura 6.2: Particionado de un dispositivo de almacenamiento



6.4 Estructura de directorios

Los sistemas operativos utilizan archivos y directorios para organizar y almacenar información en un dispositivo de almacenamiento. Los archivos contienen los datos que los usuarios crean y manipulan, mientras que los directorios proporcionan una forma de organizar y acceder a los archivos de manera lógica y fácil de entender.

! Observación

Los términos carpeta y directorio tienen el mismo significado. Estos son en realidad, un tipo de archivo especial.

La jerarquía de directorios en un sistema de archivos se llama árbol de directorios. El nivel superior del árbol de directorios se llama directorio raíz, que contiene todos los demás directorios y archivos en el sistema de archivos. A medida que se desciende por el árbol de directorios, los nombres de los directorios y los archivos se combinan para formar rutas de acceso a los archivos y directorios individuales.

! Observación

En los sistemas *Linux* existe un solo directorio raíz llamado «/», en donde luego se «*montan*» el resto de los dispositivos. En los sistemas *Windows* cada partición tiene su propio directorio raíz llamado «\».

Para navegar a través de los directorios del sistema en un entorno gráfico, utilizamos un programa denominado «*administrador de archivos*». En dicho programa podemos recorrer la estructura del sistema de archivos mediante una serie de clicks de mouse en iconos, botones y menúes.

Figura 6.3: PCManFM

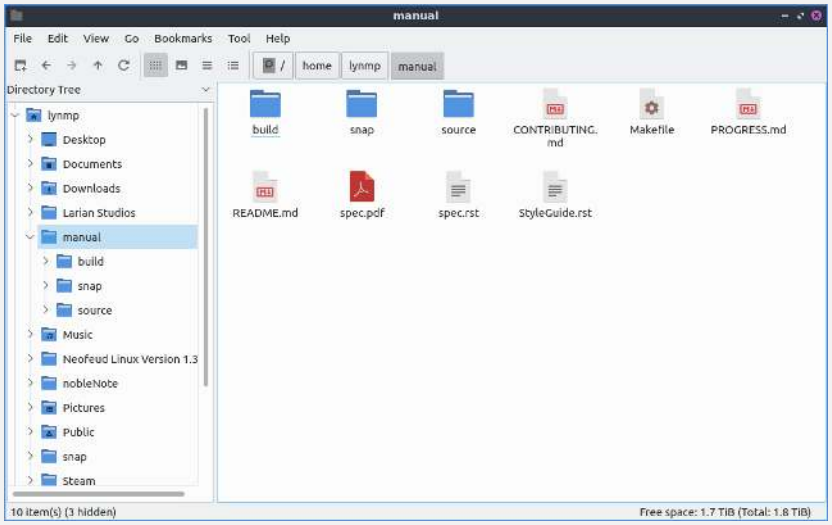
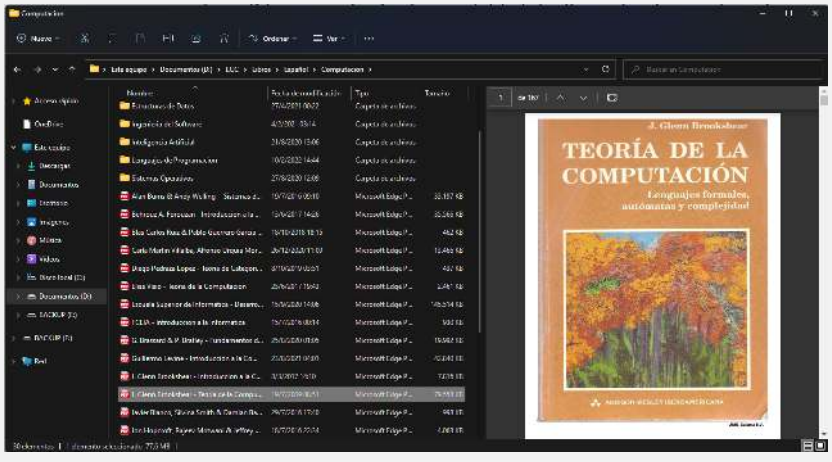


Figura 6.4: Explorador de archivos



Existen dos tipos de rutas que se utilizan para referirse a la ubicación de un archivo o directorio en un sistema de archivos: rutas absolutas y rutas relativas.

Ruta absoluta Una ruta absoluta es una ruta que especifica la ubicación completa de un archivo o directorio, desde el directorio raíz hasta el archivo o directorio en cuestión. En otras palabras, una ruta absoluta es la ruta completa que describe la ubicación exacta de un archivo o directorio, independientemente de la ubicación del usuario o del directorio de trabajo actual.

> **!** Observación
>
> Es fácil reconocer a una ruta absoluta, pues siempre comienza con «/».

Ejemplo 6.1. La ruta absoluta donde se ubica el interprete de linea de comandos bash suele ser: «`/usr/bin/bash`».

Ruta relativa Una ruta relativa es una ruta que describe la ubicación de un archivo o directorio en relación con el directorio de trabajo actual. En otras palabras, una ruta relativa es una ruta que se especifica desde el directorio actual hasta el archivo o directorio en cuestión.

Ejemplo 6.2. Si actualmente nos encontramos en «`/usr`», la ruta relativa hacia bash es: «`bin/bash`».

6.5 El sistema de archivos de Linux

6.5.1 Esquema de particionado

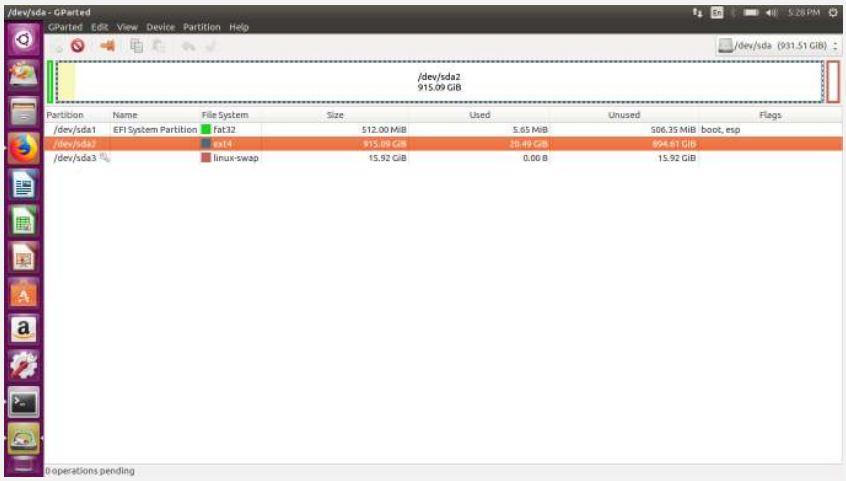
Linux puede instalarse en una variedad de esquemas de particionado, pero uno de los más comunes es el esquema de particionado convencional, que utiliza tres particiones para el sistema:

- Una partición en donde se instala el gestor de arranque y el núcleo del sistema operativo.
- Una partición destinada a almacenar los programas de usuario y documentos.
- Una partición de intercambio, reservada para ser utilizada cuando la memoria RAM se agota.

! Observación

Los esquemas de particionado mas antiguos no instalan el cargador de arranque en una partición separada sino en el primer sector del disco.

Figura 6.5: GParted



6.5.2 *Nomenclatura de dispositivos*

En *Linux*, los discos duros y particiones se nombran usando un esquema de nomenclatura estandarizado. Cada dispositivo se identifica por un nombre que se genera automáticamente y que puede estar compuesto de diferentes elementos.

El nombre asignado a un dispositivo de almacenamiento consta de tres partes:

- Una sigla identificadora del tipo de dispositivo. En general, los discos rígidos, unidades de estado sólido y dispositivos USB empiezan con las letras «`sd`».
- Una letra secuencial que identifica el dispositivo en relación con otros dispositivos del mismo tipo. El primer dispositivo se denominará «`sda`», el segundo «`sdb`» y así sucesivamente.
- Un número adicional que indica la partición específica del dispositivo. La primera partición en el primer dispositivo será «`sda1`», la segunda será «`sda2`», y así sucesivamente.

! Observación

Es muy importante notar que la denominación «`sda`» hace referencia a todo el dispositivo, mientras que «`sda1`» solo hace referencia a la primer partición del primer dispositivo.

6.5.3 Jerarquía de archivos

La jerarquía de archivos en el sistema de archivos se organiza en un árbol con una estructura específica. El objetivo de una estructura estandarizada es que los desarrolladores de software y los administradores de sistemas puedan predecir con precisión dónde se encuentran los archivos importantes del sistema y cómo acceder a ellos.

A continuación, se presentan algunos de los directorios más importantes de la jerarquía de archivos:

/ Es la raíz del sistema de archivos. Cualquier otro directorio debe encontrarse dentro de el, independientemente de en cual dispositivo se encuentre.

/bin Este directorio contiene los programas del sistema, como comandos básicos de GNU/*Linux*.

/boot Contiene los archivos necesarios para el proceso de arranque del sistema, como el gestor de arranque o el núcleo.

/dev En *Linux*, los dispositivos del hardware se representan como archivos especiales. En esta carpeta se encuentran esos archivos.

/etc Este directorio contiene los archivos de configuración del sistema y de las aplicaciones instaladas en el mismo.

/home Este directorio se utiliza para almacenar los directorios personales de los usuarios.

/lib Contiene las bibliotecas compartidas necesarias para el funcionamiento del sistema.

/mnt Este directorio se utiliza para montar sistemas de archivos externos, como unidades flash USB, discos duros externos, etc. En algunas distribuciones esto también se hace en el directorio «`/media`».

/proc El directorio «`/proc`» es un directorio virtual que contiene información sobre el sistema y los procesos en ejecución. Es generado en tiempo real por el kernel del sistema operativo.

/root Este directorio es el directorio personal del usuario «*root*». El usuario *root* es el único que tiene acceso completo al sistema.

/sbin Contiene los programas utilizados principalmente para la administración del sistema.

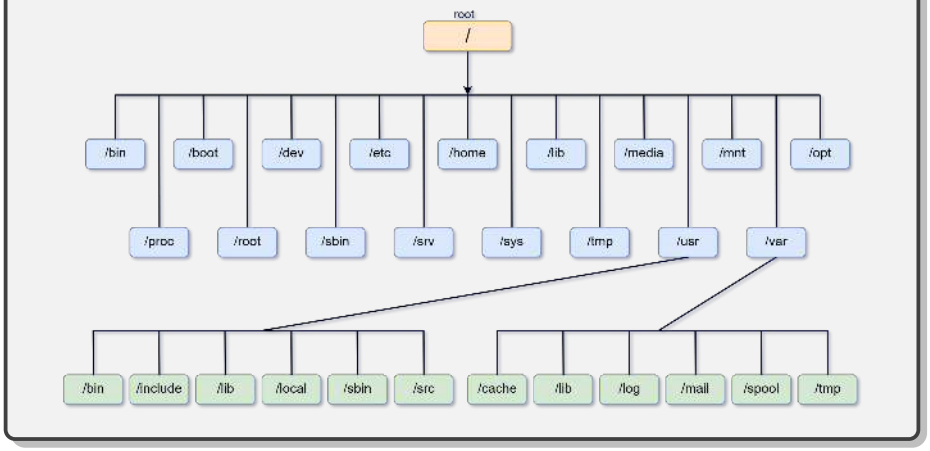
/tmp Este directorio se utiliza para almacenar archivos temporales creados por el sistema y los usuarios.

/usr Contiene la mayoría de las utilidades y aplicaciones multiusuario, es decir, accesibles para todos los usuarios. En otras palabras, contiene los archivos compartidos, pero que no obstante son de sólo lectura.

/var Este directorio contiene archivos que cambian frecuentemente durante la operación del sistema, como archivos de registro y archivos de bases de datos.

/opt Se utiliza para almacenar «*paquetes adicionales*» o software adicional que no es parte de la distribución principal del sistema operativo.

Figura 6.6: Jerarquía de directorios estándar en *Linux*.



! Observación

A diferencia de *Windows*, *Linux* es sensible a mayúsculas y minúsculas. Esto quiere decir que «`archivo`» y «`Archivo`» son considerados dos archivos diferentes.
Además todos aquellos archivos y carpetas que en su nombre comiencen con un punto, son considerados archivos ocultos.

6.5.4 *Carpetas especiales*

Existen tres carpetas especiales en los sistemas *Linux*:

- El punto (.) es una carpeta especial que representa el directorio actual en el que se encuentra el usuario.
- Los dos puntos (..) representan el directorio padre del directorio actual.
- El símbolo de tilde (~) representa el directorio principal del usuario.

! Observación

Es importante notar que el carácter «~» no tiene ningún significado para el sistema operativo, y es trabajo del shell reemplazarlo por el directorio correspondiente. Así como esta, existen muchas otras funcionalidades que son labor del interprete de linea de comandos (no del S. O.) y serán puntualizadas en el apunte.

6.5.5 *Montaje*

«*Montar*» significa hacer que un sistema de archivos esté disponible en un punto específico del sistema de archivos raíz. En otras palabras, montar un sistema de archivos implica hacer que el sistema operativo sea consciente de la existencia de un sistema de archivos y asignarlo a un directorio específico en la jerarquía de archivos del sistema.

El punto de montaje es un directorio en el sistema de archivos raíz, que se utiliza para acceder al contenido del sistema de archivos que se va a montar. Una vez montado, el contenido del sistema de archivos se vuelve accesible en el punto de montaje especificado.

6.5.6 *Enlaces*

En *Linux*, existen dos tipos de enlaces para archivos: enlaces suaves (también conocidos como enlaces simbólicos o «*soft links*») y enlaces duros (también conocidos como enlaces físicos o «*hard links*»).

Un enlace suave es un archivo especial que apunta a otro archivo en el sistema de archivos. El enlace suave no contiene los datos del archivo al que apunta, sino simplemente la ruta del archivo. Si se elimina el archivo original, el enlace suave se vuelve inútil. Los enlaces suaves se utilizan comúnmente para hacer referencia a archivos en diferentes ubicaciones y para crear accesos directos en el sistema de archivos.

! Observación

Los enlaces suaves de *Linux* son equivalentes a los «*accesos directos*» en los sistemas *Windows*.

Un enlace duro es una entrada de directorio adicional que apunta al mismo archivo en el sistema de archivos. A diferencia de los enlaces suaves, los enlaces duros no son archivos independientes y no pueden apuntar a archivos en diferentes sistemas de archivos o particiones. Si se elimina el archivo original, el enlace duro sigue siendo válido y no se elimina hasta que todos los enlaces se eliminan.

! Observación

No es posible hacer un enlace duro hacia un directorio.

6.6 Ejercicios

Los ejercicios marcados con ✎ han sido ejercicios de examen.

Ejercicio 6.3. ✎ Identifique cuáles de las siguientes rutas son absolutas y cuáles relativas:

1. `/`
2. `/home`
3. `..`
4. `home/entornos`

Ejercicio 6.4. ✎ ¿Qué ruta representa «`../../~/..`»?

Parte II

Manejo de bash

«*Pensar no garantiza que no cometeremos errores. Pero no pensar garantiza que lo haremos*».

Leslie Lamport

7

Comandos básicos

7.1 Introducción

Bash es un interprete de linea de comandos y un lenguaje de programación de scripting utilizado principalmente en sistemas operativos tipo *Unix*. Es el shell por defecto en la mayoría de las distribuciones de *Linux*.

Permite a los usuarios interactuar con el sistema operativo mediante la entrada de comandos en un emulador de terminal. Con *Bash*, los usuarios pueden ejecutar programas, gestionar archivos y directorios, automatizar tareas y realizar una amplia variedad de operaciones de administración del sistema.

Además, también es un lenguaje de scripting que permite a los usuarios escribir programas para automatizar tareas repetitivas o complejas. Estos scripts pueden incluir una serie de comandos de *Bash* y se pueden ejecutar en una terminal o programarse para que se ejecuten automáticamente en momentos específicos.

Sintaxis La sintaxis de un comando es la estructura y el formato que se deben seguir para escribir y ejecutar correctamente un comando en un sistema operativo. Esta incluye el nombre del comando, las opciones y los argumentos necesarios, así como la forma en que deben escribirse y ordenarse.

Es importante seguir la sintaxis correcta de un comando, ya que de lo contrario, el comando puede no funcionar como se espera o puede producir errores.

Cada comando tiene su propia sintaxis única, pero en general, la mayoría sigue una estructura básica:

```
comando [opciones] [argumentos]
```

- El comando es el nombre del programa o acción que se desea ejecutar.
- Las opciones son indicadores adicionales que se agregan al comando para modificar su comportamiento. Las opciones generalmente se escriben después del comando y comienzan con un guion (-) o dos guiones (--).

- Los argumentos son valores adicionales que se pasan al comando para que actúe sobre ellos. Los argumentos generalmente se escriben después de las opciones y pueden ser nombres de archivo, rutas de directorio, nombres de usuario, números, etc.
- La mayoría de los comandos admiten la opción «`--help`» para consultar su ayuda.

! Observación

En la sintaxis de un comando, los corchetes se utilizan para indicar que un elemento es opcional. Esto significa que puede omitirse o incluirse según sea necesario.

Para ejecutar un comando, primero debe escribirse en una terimal y finalmente presionar la telca ⌨ Enter.

Ejemplo 7.1. ✔ El comando «`echo`» sirve para mostrar mensajes por la pantalla, como se ilustra a continuación:

Se sugiere probar en la terminal los ejemplos marcados con ✔ .

Bash

```
echo Hola mundo!
```

Salida en pantalla

```
Hola mundo!
```

Además de estos lineamientos generales, podemos utilizar la barra invertida («\») para dividir un comando largo en varias líneas para mejorar la legibilidad sin ejecutar el comando parcialmente en cada línea.

Ejemplo 7.2. Podemos escribir en pantalla una linea muy larga en varios renglones:

Bash

```
echo Hola \
mundo\
!
```

Salida en pantalla

```
Hola mundo!
```

Prompt El «*prompt*» es el indicador que muestra el shell para indicar que está esperando la entrada del usuario. Es una cadena de caracteres que puede contener información útil, como el nombre del usuario, el nombre del equipo, la ruta actual del directorio y otros detalles.

Ejemplo 7.3. El prompt por defecto se ve así:

Bash

```
usuario@equipo:ruta$
```

Historial Cada vez que se escribe y ejecuta un comando en *Bash*, se agrega al historial de comandos. Esto permite acceder a los comandos previamente ejecutados para volver a ejecutarlos o modificarlos en lugar de volver a escribirlos desde cero. Para poder acceder fácilmente al historial, se pueden presionar las siguientes teclas:

- ⌨ Flecha hacia arriba: Muestra el comando anterior en el historial. Si se sigue presionando, se mostrarán comandos anteriores en el historial en orden cronológico inverso.
- ⌨ Flecha hacia abajo: Análogo pero en orden cronológico.
- ⌨ Ctrl+R: Abre una búsqueda reversa en el historial de comandos. Al escribir una palabra clave o frase, se mostrarán los comandos anteriores que coinciden con la búsqueda.

Además *Bash* permite repetir comandos anteriores fácilmente:

- Cuando se utiliza «`!!`» en un comando, *Bash* lo interpreta como una referencia al último comando ejecutado. Es útil cuando deseas repetir el comando anterior rápidamente.
- El formato «`!n`» se utiliza para ejecutar el enésimo comando en el historial de comandos.

Ejemplo 7.4. ✔ Veamos como podemos reutilizar el último comando:

Se sugiere probar en la terminal los ejemplos marcados con ✔ .

Bash

```
echo Hola mundo
```

Salida en pantalla

```
Hola mundo
```

Bash

```
!!
```

Salida en pantalla

```
echo Hola mundo
Hola mundo
```

Ejemplo 7.5. También podemos repetir el penúltimo comando:

Bash

```
echo Linea 1
echo Linea 2
```

Salida en pantalla

```
Linea 1
Linea 2
```

Bash

```
!-2
```

Salida en pantalla

```
echo Linea 1
Linea 1
```

Autocompletado Cuando se escribe un comando o una ruta en el terminal y se presiona la tecla ⌨ Tab, Bash intenta completar automáticamente el comando o la ruta. Si hay una sola opción disponible, Bash la completará automáticamente, de lo contrario se mostrará una lista de opciones que coinciden con lo que se ha escrito hasta el momento. Si se sigue presionando la tecla ⌨ Tab, Bash continuará mostrando opciones adicionales.

7.1.1 man

El comando `man` es una herramienta de línea de comandos que proporciona información detallada sobre otros comandos, funciones y bibliotecas.

«man» es una abreviatura de «manual».

La sintaxis básica del comando «man» es la siguiente:

```
man [opciones] comando
```

Las opciones más comunes incluyen:

- «-f» para buscar descripciones cortas de comandos relacionados con una palabra clave.
- «-k» para buscar páginas del manual que contienen una palabra clave específica.

Para navegar dentro del programa «man», se pueden usar varias teclas moverse por la página, buscar texto y salir del programa:

- Las teclas de ⌨ flecha arriba y ⌨ abajo: se pueden utilizar para desplazarse hacia arriba y hacia abajo por la página del manual.
- La tecla «⌨ /» seguida de un texto de búsqueda: se puede utilizar para buscar una palabra o frase en la página del manual actual.
- La tecla «⌨ q»: se puede utilizar para salir del programa man y volver al terminal.

! Observación

Las páginas del manual también pueden consultarse en internet: «🔗 Linux Manpages Online»

Ejemplo 7.6. ✔ Para poder observar la documentación del comando man podemos usar el siguiente comando:

Se sugiere probar en la terminal los ejemplos marcados con ✔ .

Bash
```
man man
```

Ejemplo 7.7. Podemos consultar cuales comandos sirven para «*listar directorios*» con el siguiente comando:

Bash
```
man -k "list directory"
```

Salida en pantalla

```
dir (1)  - list directory contents
ls (1)   - list directory contents
ls (1p)  - list directory contents
vdir (1) - list directory contents
```

Ejemplo 7.8. La descripción corta del comando «`clear`» la podemos observar con el siguiente comando:

Bash

```
man -f clear
```

Salida en pantalla

```
clear (1)  - clear the terminal screen
clear (3x) - clear all or part of a curses window
```

7.1.2 `clear`

El comando `clear` es un comando que se utiliza para limpiar la pantalla del terminal, eliminando todos los comandos y resultados previos que se hayan mostrado. Al ejecutar este comando, la pantalla del terminal se restablecerá, dejando la línea de comando en la parte superior de la pantalla.

La sintaxis básica del comando `clear` es la siguiente:

```
clear [-x]
```

La opción «`-x`» puede agregarse para evitar que se borre el historial desplazamiento del (5.5) **emulador de terminal**.

Observación

El atajo de teclado ⌨ Ctrl+L es equivalente al comando `clear`.

Ejemplo 7.9. ✔ Para borrar la pantalla utiliza el siguiente comando:

Se sugiere probar en la terminal los ejemplos marcados con ✔ .

Bash

```
clear
```

7.1.3 *echo*

El comando «echo» es un comando que se utiliza para imprimir texto en la pantalla del terminal.

La sintaxis básica es la siguiente:

```
echo [opciones] texto
```

También admite varias opciones y parámetros adicionales, que permiten ajustar el formato de la salida y realizar otras tareas:

- «-n»: evita que el comando agregue un carácter de nueva línea al final de la salida. En otras palabras, la siguiente línea del terminal comenzará en la misma línea en la que termina la salida del comando.
- «-e»: activa el soporte de secuencias de escape (3.3) en la salida. Por ejemplo, se pueden utilizar secuencias de escape para agregar saltos de linea al texto.

! Observación

La utilidad del comando echo resultara mas evidente una vez que se estudien las redirecciones y los scripts.

Ejemplo 7.10. Para escribir dos palabras separadas por una linea se escribe el siguiente comando:

Bash

```
echo -e "Hola\nmundo"
```

Salida en pantalla

```
Hola
mundo
```

7.1.4 *history*

El comando history muestra una lista de los comandos que se han ejecutado anteriormente. La sintaxis básica del comando «history» es la siguiente:

```
history [-c]
```

Puede agregarse la opción «-c» para borrar el historial.

! Observación

El comando `history` no es un programa de usuario, sino una funcionalidad que provee *Bash*.

Ejemplo 7.11. Para ver los últimos comandos ingresados se usa el siguiente comando:

Bash

```
history
```

Salida en pantalla

```
1 cd ~
2 ls
3 history
```

7.1.5 Ejercicios

Los ejercicios marcados con ✔ son ejercicios recomendados.

Ejercicio 7.12. ✔ Utilice el comando «`man`» para averiguar el propósito y funcionamiento de los comandos «`date`» y «`cal`». Pruebe cada uno de ellos.

Ejercicio 7.13. Consulte la ayuda del comando «`clear`» con el comando «`man`» y con la opción «`--help`».

Los ejercicios marcados con ★ son de mayor dificultad.

Ejercicio 7.14. ★ Investigue el uso de las siguientes secuencias de escape (3.3) con el comando «`echo`»:

1. «`\n`».
2. «`\t`».
3. «`\b`».
4. «`\r`».

Ejercicio 7.15. Escriba una oración en una terminal. Determine para que sirven: ⌨ Ctrl+A, ⌨ Ctrl+E, ⌨ Ctrl+K, ⌨ Alt+D y ⌨ Ctrl+W.

Ejercicio 7.16. ★ Explique la salida en pantalla de los siguientes comandos:

Los ejercicios marcados con ★ son de mayor dificultad.

Bash

```
echo ~
echo \~
echo \\~
```

Ejercicio 7.17. ★ Averigüe para que sirven los comandos «`info` y «`help`», y cuáles son sus diferencias con «`man`».

7.2 Sistema de archivos

7.2.1 `pwd`

El comando «`pwd`» muestra la ruta completa del directorio de trabajo actual en el sistema de archivos. La sintaxis básica del comando «`pwd`» es la siguiente:

«`pwd`» son las siglas de «Print Working Directory» («mostrar directorio de trabajo»).

```
pwd [-P]
```

Puede agregarse la opción «`-P`» para mostrar el directorio sin enlaces simbólicos.

Ejemplo 7.18. ✔ Para mostrar la ruta del directorio de trabajo actual usamos:

Se sugiere probar en la terminal los ejemplos marcados con ✔ .

Bash

```
pwd
```

Salida en pantalla

```
/home/entorno
```

7.2.2 `cd`

El comando «`cd`» se utiliza para cambiar el directorio de trabajo actual en el sistema de archivos. La sintaxis básica del comando «`cd`» es la siguiente:

«`cd`» son las iniciales de «cambiar directorio».

```
cd [ruta]
```

- Si no se especifica una ruta, el comando es equivalente a «`cd ~`».
- Si se utiliza el argumento «`-`», el comando «`cd`» vuelve al directorio de trabajo anterior.

! Observación

A diferencia de los programas anteriores, «`cd`» es un comando interno del shell.

Ejemplo 7.19. ✔ Podemos ir a la carpeta raíz del sistema escribiendo lo siguiente:

Se sugiere probar en la terminal los ejemplos marcados con ✔ .

Bash

```
cd /
```

Ejemplo 7.20. Para ir al directorio que contiene la carpeta personal del usuario actual usamos el siguiente comando:

Bash

```
cd ~/..
```

7.2.3 `ls`

El comando «`ls`» se utiliza para listar los archivos y directorios en el directorio de trabajo actual o en una ubicación específica del sistema de archivos. La sintaxis básica del comando «`ls`» es simplemente:

«`ls`» es una abreviación de «listar».

```
ls [ruta]
```

Si se omite la ruta, se asume que se desea listar los archivos y directorios en el directorio de trabajo actual. Además, se pueden utilizar opciones para personalizar la salida del comando «`ls`». Algunas opciones comunes incluyen:

- «`-l`»: muestra la lista en formato largo, que incluye información detallada sobre cada archivo o directorio.
- «`-h`»: muestra el tamaño de los archivos en formato legible por humanos, como «KB» o «MB».
- «`-a`»: muestra todos los archivos y directorios, incluyendo los que comienzan con un punto.

Se sugiere probar en la terminal los ejemplos marcados con ✔ .

Ejemplo 7.21. ✔ Podemos ir a la carpeta raíz del sistema escribiendo lo siguiente:

Bash
```
cd /
```

Ejemplo 7.22. ✔ Para mostrar el contenido del directorio actual usamos el siguiente comando:

Bash
```
ls
```

Salida en pantalla
```
dsh dsh.c entorno final hola.c hola.py hola.s
```

Ejemplo 7.23. ✔ Agregando la opción «`-a`» también veremos los archivos ocultos:

Bash
```
ls -a
```

Salida en pantalla
```
./ ../ dsh dsh.c entorno final hola.c hola.py hola.s
.oculto
```

Ejemplo 7.24. ✔ Para ver información adicional, agregamos la opción «`-l`»:

Bash
```
ls -l
```

Salida en pantalla
```
-rwxr-xr-x 1 entorno wheel 15832 abr 12 19:33 dsh
-rw-r--r-- 1 entorno wheel  1130 abr 12 19:32 dsh.c
drwxr-xr-x 6 entorno wheel  4096 abr 17 13:54 entorno
drwxr-xr-x 4 entorno wheel  4096 abr 19 01:03 final
-rw-r--r-- 1 entorno wheel    60 mar 20 15:05 hola.c
-rw-r--r-- 1 entorno wheel    35 mar 28 19:06 hola.py
-rw-r--r-- 1 entorno wheel   462 mar 28 19:40 hola.s
```

En este formato de salida podemos ver: los permisos del archivo, cantidad de enlaces, usuario propietario, grupo propietario, tamaño, fecha de modificación y nombre.

Ejemplo 7.25. ✔ Para mostrar el contenido del directorio raíz podemos usar el siguiente comando:

Se sugiere probar en la terminal los ejemplos marcados con ✔ .

Bash

```
ls /
```

Salida en pantalla

```
bin boot dev etc home lib lib64 lost+found mnt opt proc
root run sbin srv sys tmp usr var
```

7.2.4 `tree`

El comando «`tree`» en sistemas *Unix* y *Linux* es una herramienta de línea de comandos que muestra la estructura de directorios y archivos en forma de árbol. Su uso básico es bastante simple, y puede proporcionar una vista jerárquica clara de la organización de archivos en un directorio y sus subdirectorios.

La sintaxis básica es:

```
tree [opciones] [directorio]
```

Algunas opciones comunes incluyen:

- «`-d`»: Muestra solo los directorios, no los archivos.
- «`-L nivel`»: Limita la profundidad del árbol al nivel especificado.
- «`-a`»: Muestra archivos y directorios ocultos.
- «`-f`»: Muestra la ruta completa de cada archivo.

Ejemplo 7.26. Si ejecutamos «`tree`» en una carpeta vacía obtendremos esta salida:

Salida en pantalla

```
.

0 directories, 0 files
```

Ejemplo 7.27. Comprobemos la jerarquía de archivos de *Linux* usando el comando «`tree`»:

Bash

```
tree -L 1 /
```

Salida en pantalla

```
/
├── bin ->  usr/bin
├── boot
├── dev
├── etc
├── home
├── lib ->  usr/lib
├── lib64 ->  usr/lib
├── lost+found
├── mnt
├── opt
├── proc
├── root
├── run
├── sbin ->  usr/bin
├── srv
├── sys
├── tmp
├── usr
└── var
20 directories, 0 files
```

7.2.5 *mkdir*

El comando «`mkdir`» se utiliza para crear nuevos directorios en el sistema de archivos.

*«**mkdir**» significa* «make directory» *(o «crear directorio» en español).*

La sintaxis básica del comando «`mkdir`» es la siguiente:

```
mkdir [opciones] directorio ...
```

Donde «*opciones*» son las opciones que se pueden utilizar para personalizar la creación de la carpeta y «*directorio*» es el nombre de la carpeta que se desea crear.

Algunas de las opciones comunes incluyen:

- «`-p`»: crea todos los directorios en la ruta especificada, incluso si no existen. Si ya existe un directorio con el mismo nombre que uno de los directorios de la ruta, se ignora sin dar errores.
- «`-v`»: muestra los detalles de la creación del directorio en la salida estándar de la terminal.

Ejemplo 7.28. ✔ Para crear un directorio llamado «`carpeta`» usamos el siguiente comando:

Se sugiere probar en la terminal los ejemplos marcados con ✔ .

Bash
```
mkdir carpeta
```

Ejemplo 7.29. ✔ Para crear un directorio llamado «`carpeta1`» y otro llamdo «`carpeta2`» podemos usar este comando:

Bash
```
mkdir carpeta1 carpeta2
```

Ejemplo 7.30. ✔ Para crear un directorio llamado «`carpeta1`» con una carpeta «`carpeta2`» adentro, usamos el siguiente comando:

Bash
```
mkdir -p carpeta1/carpeta2
```

! Observación

Puede verificar que las carpetas hayan sido creadas correctamente con el comando «`ls`».

7.2.6 *`rmdir`*

El comando «`rmdir`» es utilizado para eliminar directorios **vacíos** en el sistema de archivos. La sintaxis básica es la siguiente:

«`rmdir`» es la abreviación de «remover directorio».

```
rmdir [opciones] directorio...
```

Algunas de las opciones comunes del comando «`rmdir`» incluyen:

- «`-p`»: dada una ruta, elimina un directorio y sus predecesores.
- «`-v`»: muestra los detalles de la eliminación del directorio en la salida estándar de la terminal.

! Observación

Es importante tener en cuenta que solo se pueden eliminar directorios vacíos con el comando «`rmdir`».

7.2.7 *touch*

El comando «`touch`» se utiliza para crear archivos vacíos o actualizar la fecha y hora de acceso o modificación de un archivo. La sintaxis básica del comando es la siguiente:

```
touch [opciones] archivo...
```

Algunas de las opciones comunes del comando incluyen:

- «`-a`»: actualiza sólo la fecha y hora de acceso del archivo.
- «`-m`»: actualiza sólo la fecha y hora de modificación del archivo.

Si se utiliza el comando «`touch`» para crear un archivo que no existe, se crea un archivo vacío con el nombre especificado. Si en cambio se utiliza con un archivo existente, se actualiza la fecha y hora de acceso y/o modificación del archivo sin cambiar su contenido.

! Observación

El comando «`touch`» es útil en situaciones en las que se necesita establecer manualmente la fecha y hora de acceso o modificación de un archivo, o para crear rápidamente un archivo vacío sin tener que abrir un editor de texto y guardar un archivo vacío.

Ejemplo 7.31. ✔ Podemos crear tres archivos vacíos con el siguiente comando:

Se sugiere probar en la terminal los ejemplos marcados con ✔ .

Bash

```
touch archivo1 archivo2 archivo3
```

7.2.8 *rm*

El comando «rm» es utilizado para eliminar archivos y directorios en el sistema de archivos. La sintaxis básica del comando «rm» es la siguiente:

«rm» es la abreviación de «remover».

```
rm [opciones] archivo ...
```

Algunas de las opciones comunes del comando «rm» incluyen:

- «-r»: se utiliza para eliminar directorios y su contenido de forma recursiva.
- «-f»: se utiliza para forzar la eliminación de un archivo si este no tiene permiso de escritura. Además no da error si los archivos no existen.
- «-i»: pide una confirmación antes de borrar cada archivo.

Precaución

Si se utiliza el comando «rm» para eliminar un archivo, el archivo se eliminará de forma permanente del sistema de archivos; no se mueve a una papelera de reciclaje.

Ejemplo 7.32. Para borrar tres archivos pidiendo confirmación podemos usar el siguiente comando:

Bash

```
rm -i archivo1 archivo2 archivo3
```

Salida en pantalla

```
rm: ¿borrar el fichero regular vacío 'archivo1'? (s/n) s
rm: ¿borrar el fichero regular vacío 'archivo2'? (s/n) s
rm: ¿borrar el fichero regular vacío 'archivo3'? (s/n) s
```

Ejemplo 7.33. Para borrar un directorio no vacío junto con todo su contenido usamos el siguiente comando:

Bash

```
rm -r directorio
```

7.2.9 *cp*

El comando «cp» que se utiliza para copiar uno o más archivos o directorios de una ubicación a otra en el sistema de archivos.

«cp» es una abreviación de «copiar».

La sintaxis básica es la siguiente:

```
cp [opciones] origen destino
```

Precacución

Es importante tener en cuenta que si se copia un archivo a un directorio que ya contiene un archivo con el mismo nombre, el archivo existente se sobrescribirá automáticamente sin pedir confirmación.

Algunas de las opciones comunes del comando «cp» incluyen:

- «-r»: indica que se deben copiar los directorios y su contenido de forma recursiva.
- «-v»: muestra los detalles de la copia en la salida estándar de la terminal.
- «-i»: pide confirmación antes de sobrescribir archivos existentes.

Ejemplo 7.34. Para copiar un archivo a otro directorio usamos el siguiente comando:

Bash
```
cp archivo directorio
```

Ejemplo 7.35. Para crear una copia de un archivo en el directorio actual podemos usar el siguiente comando:

Bash
```
cp archivo copia
```

Ejemplo 7.36. Para copiar un directorio y su contenido a otro directorio:

Bash
```
cp -r directorio1 directorio2
```

7.2.10 *mv*

El comando «mv» se utiliza para mover o renombrar archivos o directorios. Su sintaxis básica es la siguiente:

«mv» es una abreviatura de «mover».

```
mv [opciones] origen destino
```

El comando «mv» también ofrece algunas opciones útiles, entre las que se incluyen:

- «-i»: Solicita confirmación antes de sobrescribir archivos existentes.
- «-f»: Sobrescribe archivos existentes sin preguntar.
- «-v»: Muestra información detallada sobre lo que se está haciendo.

! Observación

Si `origen` y `destino` se encuentran dentro de la misma ruta, entonces el comando «mv» cambia el nombre del archivo o directorio.

Ejemplo 7.37. ✔ Podemos cambiar el nombre de un archivo utilizando el comando «mv»:

Se sugiere probar en la terminal los ejemplos marcados con ✔ .

Bash
```
mv nombre_original nombre_nuevo
```

Ejemplo 7.38. Para traer a la carpeta actual un archivo de otra ubicación, podemos usar el siguiente comando:

Bash
```
mv ubicacion/archivo .
```

7.2.11 *df*

El comando «df» sirve para mostrar información sobre el espacio libre y utilizado en sistemas de archivos montados. Su sintaxis básica es la siguiente:

«df» es una abreviatura de "Disk Free" (disco libre en inglés)

```
df [opciones] [ruta]
```

El comando «df» también ofrece algunas opciones útiles, entre las que se incluyen:

- «-h»: Muestra los tamaños de bloque en un formato legible por humanos (por ejemplo, en *kilobytes*, *megabytes*, *gigabytes*, etc.).
- «-T»: Muestra el tipo de sistema de archivos.

! Observación

Por defecto, si se ejecuta sin opciones, «df» muestra información sobre todos los sistemas de archivos montados en el sistema, incluyendo el espacio total, el espacio utilizado, el espacio libre y el porcentaje de uso.

Ejemplo 7.39. Para ver el tipo de sistema de archivos y espacio libre en forma legible, usamos el siguiente comando:

Bash

```
df -Th
```

Salida en pantalla

```
S.ficheros Tipo     Tamaño Usados Disp Uso% Montado en
dev        devtmpfs   971M      0 971M   0% /dev
run        tmpfs      979M   644K 978M   1% /run
/dev/sda2  ext4       9,7G   7,1G 2,1G  78% /
tmpfs      tmpfs      979M      0 979M   0% /dev/shm
tmpfs      tmpfs      979M    72M 907M   8% /tmp
/dev/sda1  vfat       100M    97M 2,9M  98% /boot
tmpfs      tmpfs      196M    16K 196M   1% /run/user/1000
```

7.2.12 du

El comando «du» sirve para mostrar información sobre el espacio utilizado por directorios. Su sintaxis básica es la siguiente:

«du» es una abreviatura de «Disk Usage» (uso de disco en inglés)

```
du [opciones] [ruta]
```

Algunas opciones útiles son:

- «`-h`»: Muestra los tamaños de archivo en un formato legible por humanos.
- «`-s`»: Muestra sólo el tamaño total de la ruta especificada, sin detalles de tamaño por directorio.
- «`-a`»: Muestra información de tamaño también para los archivos, no sólo para los directorios.

! Observación

Por defecto, si se ejecuta sin opciones, «`du`» muestra la cantidad de espacio utilizado por cada directorio en la ruta actual.

Ejemplo 7.40. Para ver el uso del directorio actual escribimos:

Bash

```
du
```

Salida en pantalla

```
4  ./semi1/cuartos1
4  ./semi1/cuartos2
12 ./semi1
4  ./semi2/cuartos4
4  ./semi2/cuartos3
12 ./semi2
28 .
```

7.2.13 `ln`

El comando «`ln`» se utiliza en sistemas *Linux* para crear enlaces simbólicos o duros (6.5.6) a archivos y directorios. Su sintaxis básica es la siguiente:

«`ln`» es una abreviatura de «link».

```
ln [-s] [archivo original] [nombre del enlace]
```

! Observación

Por defecto «`ln`» crea enlaces duros, a no ser que se utilice la opción «`-s`».

Ejemplo 7.41. Para crear un enlace suave llamado «`enlace`» que apunte al archivo «`archivo`», se puede usar el siguiente comando:

Bash

```
ln -s archivo enlace
```

7.2.14 `lsblk`

El comando «`lsblk`» es utilizado para listar información sobre discos duros, particiones y dispositivos de almacenamiento extraíbles.

El nombre «`lsblk`» proviene de «list block devices» *(«listar dispositivos de bloque»).*

La sintaxis básica de `lsblk` es:

```
lsblk [opciones]
```

Algunas opciones comunes incluyen:

- «`-a`»: Muestra todos los dispositivos disponibles.
- «`-l`»: Muestra solo una lista simple, sin detalles adicionales

Ejemplo 7.42. Observemos una posible salida del comando «`lsblk`».

Salida en pantalla

```
NAME   MAJ:MIN RM   SIZE RO TYPE MOUNTPOINTS
sda      8:0    0    10G  0 disk
+-sda1   8:1    0   100M  0 part /boot
+-sda2   8:2    0   9,9G  0 part /
sdb      8:16   0   7,5G  0 disk
+-sdb1   8:17   0   7,4G  0 part /mnt
+-sdb2   8:20   0   100M  0 part
```

Podemos observar que este sistema tiene dos unidades de almacenamiento (`sda` y `sdb`), cada una de ellas con dos particiones.

- `sda`: Esta unidad de almacenamiento tiene una capacidad de 10GB y se divide en dos particiones: `sda1` y `sda2`.
  - `sda1`: Esta es la partición de arranque y tiene una capacidad de 100MB.
  - `sda2`: Esta es la partición principal y esta montada en la raíz del sistema de archivos.

- `sdb`: Esta unidad de almacenamiento tiene una capacidad de 7,5GB y se divide en dos particiones: `sdb1` y `sdb2`.
  - `sdb1`: Esta partición se encuentra montada en la carpeta «`/mnt`».
  - `sdb2`: La partición `sdb2` no se encuentra montada en el sistema de archivos.

7.2.15 *mount*

El comando «`mount`» se utiliza en sistemas operativos tipo Linux para montar (6.5.5) sistemas de archivos en un directorio del sistema de archivos.

Su sintaxis básica es la siguiente:

```
mount [-r] dispositivo directorio
```

! Observación

Por defecto «`mount`» monta los sistemas de archivos en modo de lectura y escritura, a no ser que se use la opción «`-r`» (sólo lectura).
También es importante destacar que para montar un sistema de archivos, normalmente se necesitan privilegios de superusuario, por lo que se debe anteponer el comando «`sudo`» para ejecutar el comando «`mount`».

Ejemplo 7.43. Si se desea montar la partición «`/dev/sda1`» en el directorio «`/mnt`», se puede utilizar el siguiente comando:

Bash

```
sudo mount /dev/sda1 /mnt
```

Una vez que se ha montado la partición, cualquier archivo que se encuentre en la partición estará disponible en el directorio «`/mnt`».

7.2.16 *umount*

El comando «`umount`» se utiliza para desmontar un sistema de archivos previamente montado en un directorio. Su sintaxis básica es la siguiente:

```
umount [opciones] directorio
```

Algunas opciones útiles son las que siguen:

- «`-f`»: Forzar el desmontaje, aunque la partición esté siendo utilizada por otros procesos. El sistema puede quedar inconsistente.
- «`-l`»: Desmonta la partición permitiendo que los procesos que utilizan la partición terminen antes de desmontarla.

! Observación

Es importante destacar que para desmontar un sistema de archivos, normalmente se necesitan privilegios de superusuario, por lo que se debe anteponer el comando «`sudo`» para ejecutar el comando «`umount`».

Ejemplo 7.44. Si se desea desmontar la partición que previamente se montó en el directorio «`/mnt`», se puede utilizar el siguiente comando:

Bash

```
sudo umount /mnt
```

7.2.17 find

El comando «`find`» es una herramienta muy útil para buscar archivos y directorios en el sistema de archivos. Su sintaxis básica es la siguiente:

```
find [ruta inicial] [expresiones de busqueda]
```

Algunas de las expresiones de búsqueda mas importantes son:

- «`-type`»: Especifica el tipo de archivo que se está buscando.
- «`-name`»: Especifica el nombre del archivo que se está buscando.
- «`-empty`»: Busca archivos vacíos.

! Observación

Si no se especifica una ruta de inicio, se buscará a partir del directorio de trabajo actual.

Es posible ejecutar un comando para cada uno de los archivos o directorios encontrados. Para esto debemos agregar al final:

```
find [ruta] [expresiones de busqueda] -exec comando {} \;
```

donde «{}» se reemplazará por los nombres de los archivos o directorios encontrados.

Ejemplo 7.45. Para buscar en el directorio de trabajo actual todos los *directorios* que existan se usa el comando:

Bash

```
find -type d
```

Salida en pantalla

```
.
./semi1
./semi1/cuartos1
./semi1/cuartos2
./semi2
./semi2/cuartos4
./semi2/cuartos3
```

Ejemplo 7.46. Para buscar en el directorio de trabajo actual todos los *archivos* que existan se usa el comando:

Bash

```
find -type f
```

Salida en pantalla

```
./francia
./semi1/croacia
./semi1/cuartos1/holanda
./semi1/cuartos1/argentina
./semi1/argentina
./semi1/cuartos2/croacia
./semi1/cuartos2/brasil
./argentina
./semi2/cuartos4/marruecos
./semi2/cuartos4/portugal
./semi2/marruecos
./semi2/francia
./semi2/cuartos3/inglaterra
./semi2/cuartos3/francia
```

Ejemplo 7.47. Para actualizar la fecha de modificación y accesos de todos los archivos vacíos puede usarse el siguiente comando:

Bash

```
find -type f -empty -exec touch {} \;
```

7.2.18 Expansiones

Las expansiones, también conocidas como *comodines*, son caracteres especiales que se utilizan para representar patrones de búsqueda en **nombres de archivos y directorios**.

Los comodines pueden ser muy útiles para simplificar tareas que involucran la manipulación de muchos archivos y directorios.

! Observación

La expansión de comodines, es otra de las tareas que llevan a cabo los interpretes de línea de comandos.

- «`*`»: Representa cualquier cadena de caracteres, incluyendo una vacía.
- «`?`»: Representa cualquier carácter individual.
- «`[]`»: Representa un conjunto de caracteres posibles.
- «`{}`»: Representa un producto cartesiano de caracteres. No se tienen en cuenta los archivos de la carpeta actual.

Ejemplo 7.48. Para listar todos los archivos que terminen en «`.txt`» se puede usar el comando:

```bash
ls *.txt
```

Ejemplo 7.49. Para mostrar todos los programas de usuario que tienen «`64`» en su nombre usamos el siguiente comando:

```bash
ls /bin/*64*
```

Ejemplo 7.50. Para mostrar las segundas particiones de todas las unidades de almacenamiento podemos usar:

```bash
ls /dev/sd?2
```

Ejemplo 7.51. Para ver cuando ocupan los archivos que comienzan con dos letras y terminan con un número usamos el siguiente comando:

```bash
du [a-z][a-z][0-9]
```

Ejemplo 7.52. Para borrar todos los archivos de una sola letra, que terminen en «`.jpg`» o «`.gif`» escribimos:

```bash
rm [a-z].{gif,jpg}
```

Ejemplo 7.53. Para crear un archivo vacío por cada casilla de un tablero de ajedrez podemos escribir:

```bash
touch {a..h}{1..8}
```

7.2.19 *Ejercicios*

Los ejercicios marcados con ✔ son ejercicios recomendados.

Ejercicio 7.54. ✔ Cree un directorio vacío con «`mkdir`», verifique que se haya creado con «`ls`», ingrese y salga de el con «`cd`» y finalmente bórrelo con «`rmdir`».

Ejercicio 7.55. ✔ Utilice el comando «`cd`» para dirigirse a la carpeta *raiz*. Luego navegue hacia «`/usr/local/bin`» mediante ruta absoluta y relativa.

Ejercicio 7.56. ✔ Escriba la siguiente secuencia de comandos:

```bash
mkdir carpeta
touch carpeta/archivo
rmdir carpeta
```

Explique por qué se produce el error. Proponga una forma de solucionarlo.

Ejercicio 7.57. ✔ Escriba la siguiente secuencia de comandos:

```bash
touch carpeta
rmdir carpeta
```

Explique por qué se produce el error. Proponga una forma de solucionarlo.

Ejercicio 7.58. ✔ Escriba el siguiente comando:

```bash
mkdir carpeta1/carpeta2
```

Explique por qué se produce el error. Proponga una forma de solucionarlo.

Ejercicio 7.59. ✔ Escriba los siguientes comandos, y explique su funcionamiento:

```bash
mkdir carpeta1 carpeta2
mkdir "carpeta1 carpeta2"
```

Ejercicio 7.60. ✔ Cree dos archivos vacíos y una carpeta; luego copie los tres archivos dentro de la carpeta. Verifique con «`ls`» que todo haya salido bien.

Los ejercicios marcados con ✔ son ejercicios recomendados.

Ejercicio 7.61. ✔ Manipule el sistema de archivos para que el comando «`tree`» muestre la siguiente salida:

Salida en pantalla

```
.
├── argentina
├── francia
├── semi1
│   ├── argentina
│   └── croacia
└── semi2
    ├── francia
    └── marruecos
3 directories, 6 files
```

Ejercicio 7.62. ✔ Cree una copia idéntica a la estructura del ejercicio anterior en otro directorio, con un solo comando.

Ejercicio 7.63. ✔ Elimine ambas estructuras de directorios.

Ejercicio 7.64. ✔ Escriba un comando para mostrar los archivos ocultos de la carpeta actual.

Ejercicio 7.65. Considere la siguiente salida en pantalla del comando «`ls -Fa`»:

Salida en pantalla

```
./ ../ .../ ..../ ...../ ......
```

¿Cuántas carpetas hay en el directorio de trabajo actual?

Ejercicio 7.66. Utilice el comando «`man`» para aprender sobre el comando «`stat`».

Ejercicio 7.67. Considere la siguiente salida en pantalla del comando «`ls -log`»:

Salida en pantalla

```
total 4
drwxr-xr-x 2 4096 may 1 23:46 archivo
-rw-r--r-- 1    0 may 1 23:46 carpeta
```

¿Por qué da error el comando «`cd carpeta`»?

Ejercicio 7.68. Normalmente el comando «`cd cd`» da error. ¿En cuál caso esto no es así?

Ejercicio 7.69. Considere la siguiente salida en pantalla del comando «rm»:

Salida en pantalla

```
rm: ¿borrar el fichero regular vacío 'archivo' protegido
contra escritura? (s/n) s
```

¿Cuál fue el comando completo? ¿Cómo podría haberse omitido la comprobación de eliminación?

Ejercicio 7.70. Utilice el comando «man» para averiguar sobre el comando «shred».

Ejercicio 7.71. Escriba un comando para borrar archivos que terminen en «~» o «#» y tengan un punto «.» en su nombre.

Ejercicio 7.72. Escriba un comando para *listar* todos los archivos y carpetas del directorio actual, que tengan una letra mayúscula en su nombre.

Los ejercicios marcados con ✎ han sido ejercicios de examen.

Ejercicio 7.73. Conjeture cual será la salida por pantalla de los siguientes comandos y verifíquelo:

Bash

```
cd /
cd ~
cd ..
pwd
```

Bash

```
cd /
cd ~
cd -
pwd
```

Ejercicio 7.74. ★ Averigüe para qué sirven los comandos «pushd», «popd» y «dirs».

Ejercicio 7.75. Investigue con el comando «man», que otras opciones tiene el comando «ls». Pruebe algunas de ellas.

Los ejercicios marcados con ★ son de mayor dificultad.

Ejercicio 7.76. Escriba un comando para hacer una lista de los programas del sistema.

! Observación

Recuerde en que carpeta del sistema se ubican los programas.

Ejercicio 7.77. Investigue con el comando «`man`», las opciones «`-b`» y «`-u`» del comando «`cp`». Pruebe ambas opciones.

Ejercicio 7.78. Asumiendo que todos los meses tienen 31 días, cree un archivo vacío por cada fecha del año.

Ejercicio 7.79. Escriba la siguiente secuencia de comandos:

Bash
```
mkdir carpeta
touch carpeta/archivo1 carpeta/.archivo2
rm -rf carpeta/*
rmdir carpeta
```

Reflexione sobre el error.

Ejercicio 7.80. En una carpeta vacía, ejecute el siguiente comando y explique su resultado:

Bash
```
touch [1-9]
```

Ejercicio 7.81. En una carpeta vacía, ejecute los siguientes comandos y explique su resultado:

Bash
```
touch archivo1 archivo9
rm archivo{1..9}
```

Proponga una solución para evitar los errores.

Los ejercicios marcados con ✎ han sido ejercicios de examen.

Ejercicio 7.82. ✎ ¿Qué comando se puede usar para crear 999 carpetas?

Ejercicio 7.83. ⚠ Describa cual es el comportamiento del siguiente comando. No lo ejecute.

Los ejercicios marcados con ⚠ son ejercicios que comprometen la seguridad del equipo. Tome las precauciones necesarias.

⚠ Precaución
```
rm * .txt
```

Ejercicio 7.84. ✔ Explique la diferencia entre las diferentes salidas del comando «`pwd`»:

Los ejercicios marcados con ✔ son ejercicios recomendados.

Bash

```
cd /bin
pwd
pwd -P
```

Ejercicio 7.85. Cree enlaces simbólicos a la carpeta padre y la carpeta actual, llamados respectivamente «`padre`» y «`actual`». Verifique su comportamiento con «`pwd`».

Ejercicio 7.86. ✔ Cree una carpeta vacía y un enlace suave hacia ella. Intente acceder a ella a través del enlace. Luego elimine la carpeta original y vuelve intentar acceder a ella a través del enlace.

Los ejercicios marcados con ✔ son ejercicios recomendados.

Ejercicio 7.87. Considere la salida del comando `ls -l /bin`:

Los ejercicios marcados con ✎ han sido ejercicios de examen.

Salida en pantalla

```
lrwxrwxrwx 1 root root 7 ene 19 14:10 /bin -> usr/bin
```

Describa con que comando se creo `/bin`.

Ejercicio 7.88. ★ Piense por que se produce el error luego de introducir estos comandos:

Los ejercicios marcados con ★ son de mayor dificultad.

Bash

```
mkdir -p ~/carpeta/destino
cd ~/carpeta
ln -s destino enlace
mv enlace ..
cd ../enlace
```

¿Cómo se podría haber creado el enlace para que no de error?

Ejercicio 7.89. Utilice el comando «`man`» para aprender sobre el comando «`readlink`».

Ejercicio 7.90. ✔ Inserte un *pendrive* en la computadora y utilice el comando «`lsblk`» para ver el nombre del dispositivo. Intente montarlo y desmontarlo.

Ejercicio 7.91. ✎ Si el programa `lsblk` muestra lo siguiente en la pantalla:

Los ejercicios marcados con ✎ han sido ejercicios de examen.

Salida en pantalla

```
NAME   MAJ:MIN RM   SIZE RO TYPE MOUNTPOINTS
sda      8:0    0    10G  0 disk
+-sda1   8:1    0   100M  0 part /boot
+-sda2   8:2    0   9,9G  0 part /
sdb      8:16   0   7,5G  0 disk
+-sdb1   8:17   0   7,4G  0 part /mnt
+-sdb2   8:20   0   2,4G  0 part
```

1. ¿Cuántas particiones hay en el sistema?
2. ¿Cuántas unidades de almacenamiento hay?
3. ¿Con qué comando se montó `sdb1`?
4. ¿En qué partición se encuentra el gestor de arranque?

Ejercicio 7.92. ✔ Escriba un comando para encontrar todos los archivos terminados en «`.txt`» dentro de la carpeta actual y sus subcarpetas; y moverlos a la carpeta «`docs`».

Los ejercicios marcados con ✔ son ejercicios recomendados.

Ejercicio 7.93. ✔ Utilice el comando «`man`» y averigüe como buscar con «`find`», los archivos modificados durante la última semana.

Ejercicio 7.94. ✎ Escriba un comando para borrar archivos y directorios vacíos en la carpeta actual y sus subcarpetas.

Ejercicio 7.95. ✎ Escriba un comando para crear un archivo llamado «`.gitkeep`» en todas las carpetas y subcarpetas vacías, del directorio actual.

Ejercicio 7.96. ★ Utilice «`find`» para crear una copia de todos los directorios vacíos del directorio de trabajo actual.

Ejercicio 7.97. ★ Escriba los siguientes comandos y explique por qué el comando «`find`» no encuentra «`archivo2.txt`»:

Los ejercicios marcados con ★ son de mayor dificultad.

Bash

```
touch archivo1.txt
mkdir carpeta{1,2}
touch carpeta1/archivo1.txt carpeta2/archivo2.txt
find -name *.txt
```

Ejercicio 7.98. ⚠ En una carpeta vacía, ejecute los siguientes comandos y explique el resultado:

Los ejercicios marcados con ⚠ son ejercicios que comprometen la seguridad del equipo. Tome las precauciones necesarias.

⚠ Precaución

```
touch archivo{.txt,.doc}
ls
find -delete -name "*.txt"
ls
```

Explique por qué se produce el error. Proponga una forma de solucionarlo.

Ejercicio 7.99. ★ Averigüe que ventaja tiene usar «+» frente a «\;» cuando «`find`» se utiliza junto con «`-exec`».

Los ejercicios marcados con ★ son de mayor dificultad.

Ejercicio 7.100. ★ Escriba los siguientes comandos en una carpeta vacía e intente describir lo que sucede.

Bash

```
touch rm
ls
* *
ls
```

Ejercicio 7.101. Averigüe para que sirve el comando «`locate`».

Ejercicio 7.102. ✔ Cree los siguientes archivos:

Los ejercicios marcados con ✔ son ejercicios recomendados.

Bash

```
touch imagen1.jpg imagen2.png foto1.jpg fotoA.png
touch dato{1,2,3,A,B,-}.txt imagen{1,2}.txt
```

Utilice las expansiones de bash junto con el comando «`ls`» para listar solamente:

1. «`imagen1.jpg`» e «`imagen2.png`».
2. «`foto1.jpg`» y «`fotoA.png`».
3. Archivos terminados en 1, 2 o 3 y sufijo «`.txt`».
4. Archivos terminados en mayúsculas y sufijo «`.txt`».
5. Archivos terminados en número o letra y sufijo «`.txt`».
6. Archivos con sufijo «`.jpg`» o «`.png`».

7.3 Contenido y filtros

En sistemas tipo *Unix*, hay tres flujos de datos principales en la terminal:

Entrada estándar Es el canal desde el que un programa recibe datos. Por defecto, la entrada estándar está conectada al teclado. Cuando un programa necesita recibir información del usuario, la lee desde la entrada estándar.

Abreviada frecuentemente como «`stdin`» («Standard Input»).

Salida estándar Este es el canal de salida predeterminado para un programa. Por defecto, la salida estándar está conectada a la pantalla de tu terminal o consola. Cuando un programa quiere mostrar resultados o información al usuario, los envía a la salida estándar, y normalmente los ves en la pantalla.

Abreviada frecuentemente como «`stdout`» («Standard Output»).

Salida de error Similar a la salida estándar, pero este canal está reservado específicamente para mensajes de error y diagnósticos. Por defecto, también se muestra en la pantalla de tu terminal junto con la salida estándar. La diferencia clave es que está diseñado para separar los mensajes de error de la salida normal del programa.

Abreviada frecuentemente como «`stderr`» («Standard Error»).

! Observación

La mayoría de los comandos estan diseñados para trabajar sobre la entrada estandar, si no se especifican argumentos.

7.3.1 `nano`

«`nano`» es un editor de texto de la línea de comandos que se utiliza para crear y modificar archivos en sistemas Linux. Su sintaxis es la siguiente:

```
nano [archivo]
```

Si no se especifica un archivo, se comenzará un documento nuevo. Una vez que se encuentra en el editor, se puede usar el teclado para escribir o editar el archivo de texto.

Para utilizar el programa se pueden usar varias combinaciones de teclas que se señalan en la parte inferior de la pantalla. Algunas de ellas son:

- ⌨ Ctrl+O: Para guardar los cambios en un archivo, se utiliza la combinación de teclas ⌨ Ctrl+O, que guarda el archivo en su ubicación actual.
- ⌨ Shift+flechas: Se puede seleccionar texto manteniendo presionada la tecla ⌨ Shift y presionando las flechas del teclado.

- ⌨ Ctrl+K: Para cortar un texto seleccionado, se utiliza la combinación de teclas ⌨ Ctrl+K.
- ⌨ Ctrl+U: Para pegar un texto seleccionado, se utiliza la combinación de teclas ⌨ Ctrl+U.
- ⌨ Alt+U: Para deshacer la última acción, usamos la combinación de teclas ⌨ Alt+U.
- ⌨ Ctrl+X: Para salir del editor, se utiliza la combinación de teclas ⌨ Ctrl+X, lo que devuelve al usuario a la línea de comandos.

! Observación

Al igual que «man» y al contrario que la mayoría de los comandos que veremos, «nano» es un programa interactivo. Esto quiere decir que después de escribir el comando, se ingresa a un programa que espera mas entrada desde el teclado.

7.3.2 *cat*

El comando «cat» es una herramienta que se utiliza para concatenar y mostrar el contenido de uno o más archivos de texto. La sintaxis básica del comando «cat» es:

«cat» es una abreviación de «concatenar».

```
cat [-n] archivo...
```

Puede agregarse la opción «-n» para enumerar las líneas emitidas.

Ejemplo 7.103. ✔ Para conocer información sobre el CPU podemos observar el contenido del archivo especial «/proc/cpuinfo» con el siguiente comando:

Se sugiere probar en la terminal los ejemplos marcados con ✔ .

Bash
```
cat /proc/cpuinfo
```

Ejemplo 7.104. Para ver información sobre la memoria podemos observar el contenido del archivo especial «/proc/meminfo» con el siguiente comando:

Bash
```
cat /proc/meminfo
```

Ejemplo 7.105. Para mostrar información sobre la línea de comando del kernel utilizada para iniciar el sistema operativo y a continuación la versión del mismo, podemos usar el siguiente comando:

Bash

```
cat /proc/cmdline /proc/version
```

7.3.3 `less`

El comando «`less`» es una utilidad en línea de comandos que se utiliza para ver el contenido de un archivo de texto de manera paginada.

El nombre del comando «`less`» deriva de «`more`» que fue el programa que se utilizaba anteriormente para este propósito.

La sintaxis básica es la siguiente:

```
less archivo
```

Una vez que se ejecuta el comando «`less`», se abrirá una vista de página en la terminal que muestra el contenido del archivo. El programa «`less`» se puede manejar igual que «`man`».

! Observación

«`less`» es otro de los programas interactivos que veremos, junto con «`man`» y «`nano`».

Podemos navegar a través del documento con las siguientes teclas:

- ⌨ Flechas: Para moverte por el archivo.
- ⌨ Barra espaciadora: Para avanzar una página.
- ⌨ g: Para ir al inicio del archivo.
- ⌨ G: Para ir al final del archivo.
- ⌨ /: Para buscar en el archivo.
- ⌨ q: Para salir.

Ejemplo 7.106. ✔ Para ver un registro de los últimos sucesos del sistema podemos paginar el archivo «`/var/log/syslog`» con el comando:

Se sugiere probar en la terminal los ejemplos marcados con ✔ .

Bash

```
less /var/log/syslog
```

7.3.4 *head*

El comando «head» se utiliza para mostrar las primeras líneas de un archivo de texto. La sintaxis básica del comando es:

```
head [-n X] archivo...
```

Puede agregarse la opción «-n X» para mostrar las primeras «X» líneas del archivo.

! Observación

Por defecto, «head» muestra las primeras 10 líneas de un archivo.

Ejemplo 7.107. Para ver las primeras cuatro líneas del archivo «/etc/fstab» usamos el siguiente comando:

Bash

```
head -n 4 /etc/fstab
```

Salida en pantalla

```
# Static information about the filesystems.
# See fstab(5) for details.

# <file system> <dir> <type> <options> <dump> <pass>
```

7.3.5 *tail*

El comando «tail» se utiliza para mostrar las últimas líneas de un archivo de texto. La sintaxis básica del comando es:

```
tail [opciones] archivo...
```

Algunas opciones comunes incluyen:

- «`-n X`»: muestra las últimas «`X`» líneas del archivo en lugar de las 10 últimas.
- «`-f`»: sigue en tiempo real los cambios que se realizan en el archivo.

! Observación

La opción «`-f`» puede ser especialmente útil para monitorear en tiempo real la actividad en un archivo que está siendo constantemente actualizado.

Ejemplo 7.108. Para monitorear los últimos sucesos del sistema podemos observar las últimas líneas del archivo «`/var/log/syslog`» con el comando:

Bash

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

7.3.6 *sort*

El comando «`sort`» es una herramienta que se utiliza para ordenar líneas de texto de un archivo de entrada. La sintaxis básica del comando es:

```
sort [opciones] archivo...
```

Algunas opciones comunes incluyen:

- «`-n`»: ordena las líneas en orden numérico en lugar de orden alfabético.
- «`-r`»: ordena las líneas en orden reverso (descendente).
- «`-m`»: fusiona archivos previamente ordenados.
- «`-R`»: en vez de ordenar, mezcla el contenido.

Ejemplo 7.109. Para ordenar alfabéticamente los shells del sistema podemos usar el comando «`sort`» sobre el archivo «`/etc/shells`»:

Bash

```
sort /etc/shells
```

Salida en pantalla

```
# /etc/shells: valid login shells
/bin/bash
/bin/dash
/bin/rbash
/bin/sh
/usr/bin/bash
/usr/bin/dash
/usr/bin/rbash
/usr/bin/sh
```

7.3.7 *uniq*

El comando «uniq» es una herramienta que se utiliza para encontrar y eliminar líneas duplicadas *consecutivas* en un archivo de entrada. La sintaxis básica del comando es:

```
uniq [opciones] archivo...
```

Algunas opciones comunes incluyen:

- «-c»: para contar el número de ocurrencias de cada línea en el archivo de entrada.
- «-d»: para mostrar sólo las líneas duplicadas.
- «-u»: para mostrar sólo las líneas únicas.

! Observación

Puesto que el comando solo funciona con lineas duplicadas consecutivas, es probable que primero se deba ordenar el archivo.

7.3.8 *strings*

El comando «strings» es una herramienta que se utiliza para imprimir las cadenas de texto legibles que se encuentran en los archivos binarios. Esto es útil para buscar información dentro de archivos binarios que no son legibles para un usuario común.

La sintaxis básica es la siguiente:

```
strings [-n MINIMO] archivo...
```

Puede agregarse la opción «-n» para especificar el *MÍNIMO* de caracteres buscado.

Ejemplo 7.110. ✔ Para buscar las cadenas de 70 caracteres o mas dentro del archivo «/bin/echo» escribimos:

Se sugiere probar en la terminal los ejemplos marcados con ✔ .

Bash

```
strings -n 70 /bin/echo
```

Salida en pantalla

```
-E disable interpretation of backslash escapes (default)
NOTE: your shell may have its own version of %s, which
usually supersedes
the version described here. Please refer to your shell's
documentation
```

7.3.9 wc

El comando «wc» se utiliza para contar el número de líneas, palabras y bytes en un archivo de texto.

«wc» es una abreviatura de «Word Count» *(contar palabras).*

Esta es su sintaxis:

```
wc [opciones] archivo...
```

A continuación, se describen las opciones más comunes:

- «-l»: cuenta el número de líneas en el archivo especificado.
- «-w»: cuenta el número de palabras en el archivo especificado.
- «-c»: cuenta el número de bytes en el archivo especificado.
- «-m»: cuenta el número de caracteres en el archivo especificado.

! Observación

Si no se especifican opciones, «wc» cuenta lineas, palabras y caracteres.

7.3.10 file

El comando «file» es útil para identificar el formato de un archivo cuando la extensión del archivo no es suficiente para determinar su tipo. La sintaxis básica del comando es:

```
file archivo...
```

Ejemplo 7.111. Podemos averiguar que formato tiene una imagen con el siguiente comando:

Bash

```
file imagen
```

Salida en pantalla

```
imagen: JPEG image data, JFIF standard 1.01, resolution
(DPI), density 300x300, segment length 16, progressive,
precision 8, 739x688, components 3
```

7.3.11 cut

El comando «cut» es una herramienta de procesamiento de texto que se utiliza para cortar secciones de líneas de texto. Permite seleccionar una o varias columnas de texto de un archivo de entrada separado por delimitadores como espacios, tabulaciones o comas.

La sintaxis básica del comando es:

```
cut opciones archivo
```

Algunas de las opciones más comunes son las siguientes:

- «`-c RANGO`»: Corta un *RANGO* de caracteres.
- «`-d DELIMITADOR`»: Especifica el *DELIMITADOR* de columna utilizado en el archivo de entrada.

- «`-f CAMPOS`»: Selecciona *CAMPOS* específicos separados por el delimitador.

Ejemplo 7.112. Podemos consultar cuales son los shells usados por los usuarios del sistema con el comando:

Bash
```
cut -d : -f 7 /etc/passwd
```

7.3.12 Expresiones regulares

Las expresiones regulares son un conjunto de caracteres especiales que se utilizan para definir *lenguajes*. Un *lenguaje* es simplemente un conjunto de palabras formado por los caracteres de un determinado *alfabeto*. A su vez, un alfabeto es nada más que un conjunto de *caracteres*.

Los conceptos básicos de las expresiones regulares son:

Caracteres literales Los caracteres literales son aquellos que se escriben directamente en la expresión regular.

Ejemplo 7.113. La expresión regular «`'h'`» define el lenguaje $\{h\}$, cuya única palabra es la letra h.

Ejemplo 7.114. El lenguaje con la palabra vacía ($\{''\}$) se define mediante la expresión regular «''».

Concatenación La concatenación de lenguajes permite definir un lenguaje nuevo, a partir de otros dos. Las palabras del nuevo lenguaje se construyen teniendo como prefijos a las palabras del primero y como sufijos a las palabras del segundo. La concatenación de lenguajes no utiliza ningún carácter especial, simplemente se escribe un lenguaje a continuación de otro.

Ejemplo 7.115. El lenguaje definido por la expresión regular «`'si'`» es la concatenación del lenguaje definido por las expresiones «`'s'`» y «`'i'`»; formando el conjunto $\{si\}$.

Union La unión de lenguajes permite definir un lenguaje con palabras de otros dos. La denotamos con el carácter «`|`».

Ejemplo 7.116. El lenguaje definido por la expresión regular «`'a|b|c'`» es el lenguaje $\{a, b, c\}$.

Asociación Es posible agrupar expresiones regulares mediante la utilización de paréntesis.

Ejemplo 7.117. La expresión regular «`'(a|b|c)(1|2|3)'`» es la concatenación de los lenguajes definidos por «`'a|b|c'`» y «`'1|2|3'`»; en definitiva es el lenguaje $\{a1, a2, a3, b1, b2, b3, c1, c2, c3\}$.

Repeticiones Es posible concatenar un lenguaje consigo mismo, para dar lugar a nuevos lenguajes:

- El operador «`?`» permite concatenar un lenguaje a lo sumo una sola vez.

  Ejemplo 7.118. El lenguaje definido por la expresión regular «`'a(1)?z'`» es el lenguaje $\{az, a1z\}$.

- El operador «`{n,m}`» permite concatenar un lenguaje entre n y m veces.

  Ejemplo 7.119. Puede definirse el lenguaje $\{aaa, aaaa, aaaaa\}$ con la expresión regular «`'a{3,5}'`».

  ! Observación

  Puede omitirse el limite inferior para repetir hasta m veces, o el limite superior para repetirse a partir de n veces.

- El operador «`+`» permite concatenar un lenguaje entre una e infinitas veces.

  Ejemplo 7.120. El lenguaje definido por la expresión regular «`'x+'`» es el lenguaje $\{x, xx, xxx, xxxx, \ldots\}$.

- El operador «`*`» permite concatenar un lenguaje entre cero e infinitas veces.

  Ejemplo 7.121. El lenguaje definido por la expresión regular «`'x*'`» es el lenguaje $\{'', x, xx, xxx, xxxx, \ldots\}$.

Anclas Las anclas son caracteres especiales que indican el inicio o el final de una linea.

- El ancla «`^`» se utiliza para indicar el inicio de una linea.
- El ancla «`$`» indica el final de una linea.

! Observación

La utilidad de las anclas quedará en evidencia cuando las utilicemos para buscar contenido dentro de archivos.

Clases Ademas, contamos con algunos lenguajes ya definidos, como por ejemplo:

- La clase «`[[:digit:]]`» representa cualquier carácter numérico.

  Ejemplo 7.122. La expresión regular «`[[::digit::]]`» define el lenguaje $\{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$.

- La clase «`[[:alpha:]]`» representa cualquier carácter alfabético.
- La clase «`[[:alnum:]]`» representa cualquier carácter alfanumérico.
- La clase «`[[:lower:]]`» representa cualquier carácter en minúscula.
- La clase «`[[:upper:]]`» representa cualquier carácter en mayúscula.
- La clase «`[[:space:]]`» representa cualquier carácter en blanco.
- La clase «`[d-r]`» representa cualquier carácter entre la letra «`d`» y la «`r`».

  Ejemplo 7.123. La expresión regular «`[a-c]`» define el lenguaje $\{a, b, c\}$.

- La clase «`.`» representa cualquier carácter.

! Observación

La página web «regexr» puede utilizarse para experimentar, construir y comprender expresiones regulares. Es de mucha ayuda para comprender su comportamiento.
Además, el modulo «`exrex`» de Python puede utilizarse para mostrar el lenguaje generado por expresiones regulares. Puede instalarse con el comando «`pip install exrex`».

Ejemplo 7.124. A continuación se muestra como se pueden usar las expresiones regulares para encontrar palabras en un texto:

Archivo

```
texto
examen
setiembre
aprendiz
que
panqueque
leer
saxo
hable
panes
```

Cuadro 7.1: Resumen de expresiones regulares

Nombre	Caracteres	Ejemplo	Salida
Literales	Todos	«x»	texto examen saxo
Concatenación	Todos	«xt»	texto
Unión*	|	«i|z»	setiembre aprendiz
Asociación*	()	«x(t|a)»	texto examen
Repetición*	?	«sep?tiembre»	setiembre
Repetición*	{}	«e{2}»	leer
Repetición*	+	«(que)+»	que panqueque
Repetición*	*	«pan(que)*»	panqueque panes
Ancla	^	«^q»	que
Ancla	$	«n$»	examen
Clase	.	«b.e»	setiembre hable
Clase	[]	«[w-z]»	texto examen aprendiz saxo

7.3.13 *tr*

El comando «tr» es una herramienta para modificar caracteres en una cadena de texto. El comando toma la entrada estándar y la traduce o modifica de acuerdo con los parámetros especificados, enviando el resultado a la salida estándar.

«tr» es una abreviación de «traducir».

Se puede utilizar para realizar diversas operaciones de manipulación de texto, como eliminar o reemplazar caracteres, cambiar mayúsculas y minúsculas, y eliminar espacios en blanco. También pueden usarse clases de caracteres.

La sintaxis básica es la siguiente:

```
tr [opciones] CADENA1 [CADENA2]
```

Algunas de las opciones mas comunes son:

- «`-d`»: sirve para eliminar los caracteres especificados en el conjunto de caracteres a traducir.
- «`-s`»: sustituye secuencias de caracteres repetidos, por uno solo de ellos.

! Observación

La utilidad de «`tr`» resultara mas evidente cuando se estudien las redirecciones.

7.3.14 grep

El comando «`grep`» se utiliza para buscar patrones de texto en el contenido de archivos. La sintaxis básica del comando es la siguiente:

«grep» son las iniciales de «Global Regular Expresion Print».

```
grep [opciones] patrón archivo
```

Algunas opciones comunes son:

- «`-i`»: busca el patrón de forma insensible a mayúsculas y minúsculas.
- «`-v`»: muestra las líneas que no contienen el patrón.
- «`-c`»: muestra el número de líneas que contienen el patrón.
- «`-n`»: muestra el número de línea de cada línea que contiene el patrón.
- «`-r`»: busca recursivamente en todos los archivos dentro de un directorio y sus subdirectorios.
- «`-E`»: habilita el uso de expresiones regulares extendidas. Las expresiones regulares extendidas permiten caracteres especiales como «`?`», «`+`», «`{}`», «`|`» y «`()`».
- «`-o`»: muestra sólo la parte de la línea que coincide con el patrón de búsqueda.

- «-w»: realiza una búsqueda de palabras completas.

> **!** Observación
>
> Por defecto, «grep» muestra toda la línea que contiene la coincidencia.

Ejemplo 7.125. Para buscar cuantas veces aparece la palabra «GNU» en la documentación de Bash usamos el siguiente comando:

Bash

```
grep -c GNU /usr/share/doc/bash/*
```

Salida en pantalla

```
/usr/share/doc/bash/bash.html:9
/usr/share/doc/bash/bashref.html:49
/usr/share/doc/bash/CHANGES:18
/usr/share/doc/bash/COMPAT:0
/usr/share/doc/bash/FAQ:10
/usr/share/doc/bash/INTRO:1
/usr/share/doc/bash/NEWS:10
/usr/share/doc/bash/POSIX:0
/usr/share/doc/bash/RBASH:0
/usr/share/doc/bash/README:2
```

Ejemplo 7.126. Para buscar la linea donde se encuentra definido el usuario «root» podemos usar el siguiente comando:

Bash

```
grep root /etc/passwd
```

Salida en pantalla

```
root:x:0:0::/root:/bin/bash
```

Ejemplo 7.127. Para encontrar las lineas de un archivo donde aparece la *palabra* «if», podemos escribir el siguiente comando:

Bash

```
grep -nw if archivo
```

7.3.15 ~~sed~~

7.3.16 Ejercicios

Los ejercicios marcados con ✔ son ejercicios recomendados.

Ejercicio 7.128. ✔ Utilice «nano» para crear un archivo de texto. Luego visualice dicho contenido en el emulador de terminal con el comando «cat».

Ejercicio 7.129. ✔ Cree otro archivo diferente y observe el contenido de ambos utilizando un solo comando.

Ejercicio 7.130. Averigue para que sirve el comando «tac».

Ejercicio 7.131. En una (5.5) **consola virtual**, utilice «cat» para ver el contenido de «/usr/share/doc/git/copyright». Tome nota de las dificultades que se presentan al ver el archivo y proponga una forma de solucionarlo.

Ejercicio 7.132. Investigue como hacer que el comando «head» muestre los primeros caracteres de un archivo. Puede asumir una codificación de un byte por caracter.

Ejercicio 7.133. Utilice el comando «man» para averiguar como mostrar con «head» todas las líneas de un archivo, salvo las últimas 3. Análogamente averigue como usar «tail» para mostrar todas las lineas salvo las primeras 3.

Los ejercicios marcados con ★ son de mayor dificultad.

Ejercicio 7.134. ★ Cree un archivo vacío y, en un emulador de terminal, utilice «tail» para *monitorear* su contenido. Modifique el contenido del archivo en otra aplicación y observe como esos cambios se reflejan en el emulador de terminal.

Ejercicio 7.135. ✔ Explique la siguiente salida del comando sort y proponga una forma de solucionarla:

Bash

```
sort numeros
```

Salida en pantalla

```
1
10
2
3
4
5
6
7
8
9
```

Los ejercicios marcados con ✔ son ejercicios recomendados.

Ejercicio 7.136. ✔ Cree el siguiente archivo y explique por que el comando «`uniq archivo`» muestra palabras repetidas:

Archivo

```
hola
mundo
chau
mundo
```

Ejercicio 7.137. ✔ Descargue la siguiente fotografía y utilice el comando «`strings`» para averiguar con que cámara se tomo dicha imagen. Verifique dicha información con el comando «`file`».

Ejercicio 7.138. Utilice «`wc`» para averiguar cuantos bytes ocupa un archivo. Comparelo con la salida del comando «`ls`» con la opción apropiada.

Ejercicio 7.139. Cree un archivo con el siguiente contenido. Luego explique la salida del comando «`wc`».

Bash

```
wc -c operacion
```

Salida en pantalla

```
4 ÷ 2 = 2
```

Ejercicio 7.140. ✔

1. Cree un archivo de python con el siguiente código:

   Código

   ```
   print("Hola mundo")
   ```

2. Utilice el comando «`file`» para averiguar que tipo de archivo es.
3. Agregue la siguiente linea al comienzo del archivo:

   Código

   ```
   #!/bin/python
   ```

4. Vuelva a averiguar el tipo del archivo.

Ejercicio 7.141. Prueba el comando «`file`» en directorios, enlaces y dispositivos.

Ejercicio 7.142. ✔ Cree un archivo con el siguiente contenido:

Los ejercicios marcados con ✔ son ejercicios recomendados.

Archivo

```
Año,Marca,Modelo,Descripción,Precio
1997,Ford,E350,"ac, ABS, moon",3000.00
1999,Chevy,Venture,Extended Edition,4900.00
1999,Chevy,Venture,"Extended Edition, Very Large",5000.00
1996,Jeep,Grand Cherokee,"MUST SELL! air, moon roof",4799.00
```

Utilice el comando «`cut`» para obtener todas las marcas.

Ejercicio 7.143. Escriba un comando para extraer todos los *dominios* de un archivo cuyas lineas son correos electónicos.

Ejercicio 7.144. ★ Explique la siguiente salida por pantalla:

Los ejercicios marcados con ★ son de mayor dificultad.

Archivo

```
Nombre Apellido
Damian Ariel
Andrea Leon
Ana    Murinigo
```

Bash

```
cut -d " " -f 2 archivo
```

Salida en pantalla

```
Apellido
Ariel
Leon
```

Ejercicio 7.145. Utilice el comando «`man`» para averiguar el propósito y funcionamiento de los comandos «`comm`» y «`diff`». Pruebe cada uno de ellos.

Ejercicio 7.146. ✔ Escriba expresiones regulares que definan los siguientes lenguajes:

1. $\{xx, yx\}$.
2. Palabras compuestas solamente por una cantidad par de letras x.
3. Palabras compuestas solamente por una cantidad impar de letras x.
4. Palabras formadas solamente por letras x, o solamente por letras y.
5. Palabras que comienzan por letras x y siguen por letras y.
6. Palabras donde cada letra y se encuentra entre un par de letras x.
7. Palabras que terminen en *.jpg* o *.jpeg*.

Los ejercicios marcados con ✎ han sido ejercicios de examen.

Ejercicio 7.147. ✎ Escriba un comando para encontrar líneas de un archivo, que *no terminen* en punto y coma «;».

Ejercicio 7.148. ✎ Escriba un comando para encontrar lineas dentro de un archivo que contengan una patente de auto. Considere como patentes válidas a un cualquier palabra que empiece por dos letras mayúsculas, siga por tres dígito (sin empezar con cero) y termine con dos letras mayúsculas.

Ejercicio 7.149. ✎ Escriba un comando para encontrar lineas dentro de un archivo que contengan un DNI. Considere como DNI válido a un número de 8 dígitos, que no comienza con ceros e incluye un guión («-») como separador de miles.

Ejercicio 7.150. ✎ Escriba un comando para encontrar lineas dentro de un archivo que contengan un nombre de persona. Considere como nombres válidos a un cualquier palabra que solo contenga letras, y que empiece y contenga una sola mayúscula.

Ejercicio 7.151. ✎ Escriba un comando que muestre los mails (*algo@algo.algo*) que haya en un archivo.

Los ejercicios marcados con ★ son de mayor dificultad.

Ejercicio 7.152. ★ Escriba un comando para encontrar los precios del siguiente archivo:

Archivo

```
Fideos: $10
Chocolate: Sin stock.
Papas fritas: $25,50
OFERTA$ E$PECIALE$
Este 31 de febrero: 2x1
```

Ejercicio 7.153. ★ Utilice el comando «`man`» para consultar que hacen las opciones «`-A`» y «`-B`» del comando «`grep`».

Ejercicio 7.154. Utilice el comando «`man`» para consultar cuál es el propósito de los programas «`nl`» y «`fold`».

7.4 Secuenciación, redirección y tuberías

7.4.1 Secuenciación

La secuenciación de comandos en Bash se utiliza para ejecutar varios comandos de forma consecutiva en una sola línea. Se pueden utilizar diferentes operadores de secuenciación para controlar cómo se ejecutan los comandos en función de sus resultados.

! Observación

Los operadores de secuenciación, son otras de las cuestiones de las cuales se encarga el shell.

- «;»: Se utiliza para ejecutar varios comandos en secuencia, independientemente del resultado de cada uno de ellos.
- «&&»: Se utiliza para ejecutar el siguiente comando solo si el comando anterior tuvo éxito.
- «||»: Se utiliza para ejecutar el siguiente comando solo si el comando anterior falló.

Ejemplo 7.155. ✔ Si queremos ejecutar dos comandos seguidos, lo hacemos de la siguiente manera:

Se sugiere probar en la terminal los ejemplos marcados con ✔ .

Bash
```
echo hola; echo mundo
```

Ejemplo 7.156. ✔ Para crear dos archivos y verificar que se hayan creado, podemos usar el siguiente comando:

Bash
```
touch archivo1; touch archivo2; ls
```

Ejemplo 7.157. ✔ Podemos crear un directorio e ingresar en el, en caso de éxito con el comando:

Bash
```
mkdir directorio && cd directorio
```

Vale la pena observar que sucede en caso de fallo:

Bash
```
mkdir . && cd /
```

Ejemplo 7.158. ✔ También podemos mostrar un mensaje en caso de error:

Bash
```
rmdir . || echo "El directorio no se ha borrado."
```

7.4.2 *Redirección*

Las redirecciones en Bash son una forma de cambiar la entrada y/o salida de un comando a diferentes archivos o dispositivos.

Hay varios tipos de redirecciones en Bash:

- Redirección de salida estándar «>»: esta redirección cambia la salida en pantalla del comando a un archivo. La sintaxis básica de la redirección de salida estándar es:

```
comando > archivo
```

> **! Observación**
>
> La redirección de salida estándar substituye por completo el contenido de un archivo.

Ejemplo 7.159. Si queremos que la salida del comando «`ls`» se guarde en el archivo «`lista.txt`», podemos usar la redirección de salida de la siguiente manera:

Bash
```
ls > lista.txt
```

- Anexación de salida estándar «>>»: La redirección «>>» en Bash se utiliza para redirigir la salida estándar de un comando a un archivo y agregarla al final del archivo en lugar de reemplazar su contenido, como hace la redirección ">". La sintaxis básica de la anexación de salida estándar es:

```
comando >> archivo
```

Ejemplo 7.160. Si queremos crear un archivo con los textos «*Hola mundo*» y «*Chau mundo*» en dos líneas, podes usar el siguiente comando:

Bash
```
echo "Hola mundo" > saludo
echo "Chau mundo" >> saludo
```

- Redirección de salida de error «2>»: Esta redirección cambia la salida de error de un comando a un archivo. La sintaxis básica de la redirección de salida estándar es:

```
comando 2> archivo
```

! Observación

La salida de error en sistemas Linux es una salida de información que se genera cuando un comando o programa encuentra un error durante su ejecución. Esta salida suele estar separada de la salida estándar; aunque se suele mostrar en la misma pantalla.

Ejemplo 7.161. Para escribir los mensajes de error de «`mkdir`» en un archivo usamos el siguiente comando:

Bash
```
mkdir . 2> error
```

- Redirección de salidas «&>»: La redirección «&>» en Bash es una forma de redireccionar tanto la salida estándar como la salida de error estándar de un comando a un archivo o dispositivo. La sintaxis básica es la siguiente:

```
comando &> archivo
```

Ejemplo 7.162. Para redirigir ambas salidas del comando «`mkdir`» en un archivo usamos el siguiente comando:

Bash
```
mkdir -v carpeta carpeta &> salidas
```

Además, en Linux y sistemas similares, la carpeta «`/dev`» contiene algunos archivos especiales que son útiles a la hora de combinarlos con redirecciones. Algunos de ellos son:

- «`/dev/null`»: un archivo que no almacena ningún dato, sino que simplemente los descarta.
- «`/dev/tty*`»: son archivos que representan a las consolas virtuales.
- «`/dev/pts/*`»: son archivos que representan a los emuladores de terminales.
- «`/dev/zero`»: un archivo infinito que contiene datos cero (0).
- «`/dev/random`»: un archivo que genera datos aleatorios.

Ejemplo 7.163. Podemos hacer una lista de todos los archivos del sistema ocultando los errores de permisos con el siguiente comando:

Bash
```
ls / -R 2> /dev/null
```

7.4.3 Tuberías

En Bash, una tubería (también conocida como «*pipe*» en inglés) permite conectar la *salida* de un comando con la *entrada* de otro, lo que permite crear una cadena de procesos que trabajan juntos para realizar una tarea más compleja.

La sintaxis de una tubería es la siguiente:

```
comando | comando
```

Ejemplo 7.164. Si queremos ver los primeros diez archivos del directorio actual podemos usar el siguiente comando:

Bash
```
ls | head
```

Ejemplo 7.165. Podemos contar cuantas veces aparece una «palabra» en un archivo utilizando tuberías de la siguiente manera:

Bash
```
cat archivo | grep palabra | wc -l
```

Ejemplo 7.166. Para hacer una lista de las palabras repetidas en un archivo podemos usar el siguiente comando:

Bash
```
cat archivo | tr " " "\n" | sort | uniq -d
```

7.4.4 tee

El comando «tee» se utiliza para leer desde la entrada estándar (generalmente datos de un comando anterior a través de una tubería) y escribir tanto en la salida estándar como en uno o más archivos especificados.

La sintaxis básica es la siguiente:

```
tee [-a] archivo1 [archivo2 ...]
```

La opción «-a» permite que «tee» agregue contenido a los archivos en lugar de sobrescribirlos.

! Observación

El comando «tee» es útil cuando necesitas mostrar o almacenar la salida de un comando en un archivo y al mismo tiempo visualizarla en la pantalla.

Ejemplo 7.167. Podemos agregar contenido a muchos archivos usando el siguiente comando:

Bash
```
echo 'print("Fin del programa.")' | tee -a *.py
```

Ejemplo 7.168. Si necesitamos ejecutar un programa y al mismo tiempo guardar su salida a modo de registro podemos usar el comando «tee»:

Bash
```
python entrenamiento.py | tee log.txt
```

7.4.5 *Ejercicios*

Ejercicio 7.169. Utilice «`man`» para averiguar sobre los comandos «`true`» y «`false`». Utilice secuenciación para mostrar un mensaje en pantalla luego de la ejecución de cada uno de ellos.

Ejercicio 7.170. ✔ Escriba un comando para crear un archivo con el contenido de *todos los archivos* del directorio actual.

Los ejercicios marcados con ✔ son ejercicios recomendados.

Ejercicio 7.171. ✔Piense que ocurre tras ejecutar el siguiente comando. Verifíquelo.

```Bash
echo 2 * 3 > 10
```

Ejercicio 7.172. ✔ Busque en Internet o consulte a alguna I. A. sobre la redirección de entrada «<». Explique que hace el siguiente comando:

Los ejercicios marcados con ✔ son ejercicios recomendados.

```Bash
cat < archivo1 > archivo2
```

Ejercicio 7.173. ¿Cómo puede utilizarse la redirección de salida estándar para crear archivos vacíos, sin usar «`touch`»?

Ejercicio 7.174. ¿Cuanto ocupa un archivo con todas las «contraseñas» posibles de cuatro letras formadas por caracteres alfabéticos en minúscula?

Ejercicio 7.175. Cree un archivo con una lista de todos los archivos del sistema que terminen en «`.jpg`». Evite mostrar errores en pantalla.

Ejercicio 7.176. ✔ Cree un archivo con muchos números. Escriba secuencias de tuberías para encontrar el máximo y el mínimo.

Los ejercicios marcados con ✔ son ejercicios recomendados.

Ejercicio 7.177. ✔Escriba una secuencia de tuberías para observar la tercer linea de un archivo.

Ejercicio 7.178. ✔ Escriba una secuencia de tuberías para averiguar cual es el archivo con mayor cantidad de líneas.

Los ejercicios marcados con ✔ son ejercicios recomendados.

Ejercicio 7.179. Escriba una secuencia de tuberías para contar cuantos directorios hay en la carpeta actual.

Ejercicio 7.180. Escriba una secuencia de tuberías para averiguar cuantos emuladores de terminal hay abiertos.

Ejercicio 7.181. Escriba una secuencia de tuberías para mostrar los tamaños de los archivos del directorio actual.

Ejercicio 7.182. Escriba una secuencia de tuberías para averiguar cuantas veces se ejecutó el comando «`cd`».

Ejercicio 7.183. Escriba una secuencia de tuberías para averiguar cuantos archivos en «`/usr/share/doc`» contienen la palabra «`GNU`».

Ejercicio 7.184. Utilice los comandos «`echo`», «`cat`» y «`tee`» para agregar contenido al comienzo de un archivo.

Ejercicio 7.185. Considere el siguiente comando:

Bash

```
a | b | c && d || e | f
```

1. ¿Cuántos programas ejecuta el shell si el programa «`c`» falla?
2. ¿Cuántos programas ejecuta el shell si los programas «`c`» y «`d`» tienen éxito?
3. ¿Cuántos programas ejecuta el shell si «`c`» tiene éxito y «`d`» falla?

8

Comandos avanzados

8.1 Usuarios, grupos y permisos

8.1.1 `whoami`

El comando «`whoami`» muestra el nombre de usuario de la cuenta actualmente activa en la sesión de terminal. Es útil cuando se necesita saber el nombre de usuario para realizar una acción o para verificar que la cuenta activa es la deseada.

El comando «`whoami`» significa «Who Am I» *(quién soy yo).*

Ejemplo 8.1. Para saber cuál es el usuario activo, simplemente escribimos:

Bash

```
whoami
```

Salida en pantalla

```
alumno
```

8.1.2 `id`

El comando «`id`» obtiene información sobre la identidad de un usuario, como su nombre de usuario, identificador de usuario (*UID*), identificador de grupo (*GID*) y los grupos a los que pertenece. La sintaxis básica del comando es la siguiente:

```
id [usuario]
```

Si no se especifica el usuario, se considera el usuario que lo ejecuta.

! Observación

Dicha información se obtiene del archivo «`/etc/passwd`».

Ejemplo 8.2. Para conocer información del usuario *root* usamos el siguiente comando:

Bash

```
id root
```

Salida en pantalla

```
uid=0(root) gid=0(root) grupos=0(root)
```

8.1.3 `who`

El comando «`who`» muestra información acerca de los usuarios que están actualmente conectados al sistema. Esta información incluye el nombre de usuario, la consola virtual en la que están conectados, la fecha y hora en que iniciaron sesión y la dirección IP de la máquina desde la que se conectaron.

Ejemplo 8.3. Para saber cuáles usuarios están conectados podemos escribir:

Bash

```
who
```

Salida en pantalla

```
damian tty1 2023-05-01 04:21
root   tty2 2023-05-03 06:01 (190.2.103.252)
```

8.1.4 `su`

El comando «`su`» es utilizado para cambiar de usuario en una sesión de terminal. La básica del comando es la siguiente:

«`su`» es un acrónimo de «substituir usuario».

```
su [USUARIO]
```

Si no se especifica un *USUARIO*, se cambia al usuario *root*.

! Observación

Al ingresar el comando «su», el sistema solicitará la contraseña *del usuario al que se quiere cambiar*. Si la contraseña es correcta se iniciará una shell para el usuario indicado.

Ejemplo 8.4. Para empezar a utilizar la terminal como el usuario root escribimos:

Bash

```
su
```

8.1.5 `sudo`

Con «sudo», un usuario común puede obtener temporariamente los permisos de un usuario con otros privilegios, como el superusuario (*root*), para realizar tareas que requieren permisos especiales. La sintaxis básica del comando «sudo» es la siguiente:

```
sudo [-u USUARIO] comando
```

Si no se especifica un *USUARIO*, se obtienen los privilegios del usuario *root*.

! Observación

Al ingresar el comando «sudo», el sistema solicitará la contraseña *del usuario que ejecuta el comando*. Si la contraseña es correcta y el usuario tiene permisos para usar «sudo», el comando especificado se ejecutará con los permisos correspondientes.

Ejemplo 8.5. Para ver el archivo de contraseñas de Linux podemos usar el siguiente comando:

Bash

```
sudo head -n 5 /etc/shadow
```

Salida en pantalla

```
root:x:0:0::/root:/bin/bash
bin:x:1:1::/:/usr/bin/nologin
daemon:x:2:2::/:/usr/bin/nologin
mail:x:8:12::/var/spool/mail:/usr/bin/nologin
ftp:x:14:11::/srv/ftp:/usr/bin/nologin
```

Para usar la consola como superusuario, pero sin saber la contraseña de *root* podemos usar:

Bash

```
sudo su
```

8.1.6 passwd

El comando «passwd» es utilizado para cambiar la contraseña de un usuario. La básica del comando es la siguiente:

```
passwd [opciones] [usuario]
```

Si se ejecuta sin argumentos, se cambiará la contraseña del usuario actual. El comando pedirá al usuario que introduzca la contraseña *actual* y, a continuación, que introduzca la nueva contraseña dos veces para confirmarla.

! Observación

Las contraseñas de los usuarios en Linux están cifradas y almacenadas en un archivo protegido llamado «/etc/shadow».

Algunas de las opciones más comunes son:

- «-l»: Bloquea la cuenta del usuario. Esto significa que el usuario no podrá iniciar sesión hasta que la cuenta sea desbloqueada.
- «-u»: Desbloquea una cuenta que ha sido bloqueada.
- «-d»: Elimina la contraseña del usuario. Esto permite que el usuario inicie sesión sin una contraseña.
- «-e»: Hace que la contraseña del usuario expire. Esto significa que el usuario deberá cambiar su contraseña la próxima vez que inicie sesión.

8.1.7 Permisos en Linux

Los permisos Linux son utilizados para proteger los archivos y directorios del sistema de accesos no autorizado. Cada archivo y directorio es propiedad de un usuario y un grupo; y el propietario del archivo o directorio puede especificar quién tiene acceso a ellos y en qué nivel.

Existen tres tipos de permisos: de usuario (*u*), de grupo (*g*) y otros (*o*), y cada uno de ellos puede tener tres permisos posibles: lectura (*r*), escritura (*w*) y ejecución (*x*).

- El permiso de lectura (*r*) permite al usuario ver el contenido de un archivo o listar los archivos de un directorio.
- El permiso de escritura (*w*) permite al usuario modificar el contenido de un archivo o crear, eliminar y renombrar archivos dentro de un directorio.
- El permiso de ejecución (*x*) permite al usuario ejecutar un archivo o entrar en un directorio.

Ejemplo 8.6. Observemos la salida del comando «`ls -l ~`»:

Salida en pantalla

```
-rwxr-xr-x 1 alumno wheel 15832 abr 12 19:33 dsh*
-rw-r--r-- 1 alumno wheel  1130 abr 12 19:32 dsh.c
```

- Podemos observar que el archivo «`dsh`» tiene los permisos «`rwxr-xr-x`». Los primeros tres permisos se corresponden con los permisos de usuario (*u*), los siguientes con los permisos del grupo (*g*) y los últimos son los permisos de los otros (*o*) usuarios. En definitiva el usuario «*alumno*» tiene todos los permisos, pero los del grupo «*wheel*» y demás usuarios no tienen permiso de escritura.
- Para el caso del archivo «`dsh.c`», la única diferencia es que nadie puede ejecutarlo.

En ocasiones estos permisos se representan mediante un número de 3 dígitos en base 8, donde cada dígito representa los permisos para un grupo diferente de usuarios: el propietario del archivo, el grupo al que pertenece el archivo y otros usuarios. Cada dígito es una suma de valores numéricos que representan los permisos para leer, escribir y ejecutar.

Los valores numéricos correspondientes a cada permiso son:

- 4 para el permiso de lectura (*r*).
- 2 para el permiso de escritura (*w*).
- 1 para el permiso de ejecución (*x*).

Por lo tanto, la suma de los valores numéricos para los distintos permisos es:

- 0: sin permisos (`---`).
- 1: permiso de ejecución (`--x`).
- 2: permiso de escritura (`-w-`).
- $1 + 2 = 3$: permiso de escritura y ejecución (`-wx`).
- 4: permiso de lectura (`r--`).
- $1 + 4 = 5$: permiso de lectura y ejecución (`r-x`).
- $2 + 4 = 6$: permiso de lectura y escritura (`rw-`).
- $1 + 2 + 4 = 7$: permiso de lectura, escritura y ejecución (`rwx`).

Ejemplo 8.7. Si vemos que un archivo tiene permisos 644, esto significa que el propietario del archivo tiene permisos de lectura y escritura (6), mientras que el grupo y otros usuarios solo tienen permiso de lectura (4).

8.1.8 `chown`

El comando «`chown`» se utiliza para cambiar el propietario y/o grupo de un archivo o directorio. La sintaxis general del comando es:

«`chown`» es una abreviación de «Change Owner» *(cambiar dueño).*

```
chown [-R] usuario[:grupo] archivo
```

Puede agregarse la opción «`-R`» para operar recursivamente sobre los directorios.

! Observación

Es importante tener en cuenta que generalmente solo el usuario *root* puede cambiar la propiedad de un archivo o directorio.

Ejemplo 8.8. El usuario *root* puede apropiarse de un «archivo» con el comando:

Bash
```
chown root archivo
```

8.1.9 *chmod*

El comando «chmod» se utiliza para cambiar los permisos de acceso a archivos y directorios. La sintaxis básica del comando «chmod» es la siguiente:

*«**chmod**» significa «cambiar modo» (del inglés: «Change Mode»).*

```
chmod [-R] modo archivo
```

Puede agregarse la opción «-R» para operar recursivamente sobre los directorios.

El modo puede especificarse en forma numérica o usando caracteres especiales. La sintaxis básica del modo simbólico es:

```
[ugoa][+-=][rwx]
```

Donde:

- «u»: representa al usuario propietario del archivo/directorio.
- «g»: representa al grupo al que pertenece el archivo/directorio.
- «o»: representa a otros usuarios que no son el propietario ni pertenecen al grupo.
- «a»: representa a todos los usuarios.
- «+»: agrega los permisos especificados al archivo/directorio.
- «-»: elimina los permisos especificados del archivo/directorio.
- «=»: establece los permisos especificados y elimina cualquier otro permiso que no sea el especificado.

Ejemplo 8.9. Para agregar permiso de *lectura* al usuario propietario de «archivo» podemos usar el comando:

Bash
```
chmod u+r archivo
```

Ejemplo 8.10. Para eliminar permiso de *escritura* al grupo al que pertenece «archivo» podemos usar el comando:

Bash
```
chmod g-w archivo
```

Ejemplo 8.11. Para establecer permiso de *ejecución* a otros usuarios que no son propietarios de «archivo» ni pertenecen al grupo, podemos usar el comando:

Bash
```
chmod o=x archivo
```

Ejemplo 8.12. Para darles todos los permisos a todos los usuarios sobre «archivo» podemos escribir:

Bash
```
chmod =777 archivo
```

8.1.10 useradd

El comando «useradd» es utilizado para crear una nueva cuenta de usuario en el sistema. A continuación se explican en detalle los aspectos más importantes del comando.

Sintaxis básica:

```
useradd [opciones] usuario
```

Algunas de las opciones más comunes son:

- «-m»: Crea automáticamente el directorio personal del usuario en «/home/usuario».
- «-g GRUPO»: Setea el *GRUPO* al que pertenece el usuario.

! Observación

Es importante destacar que para crear una cuenta de usuario con «useradd», es necesario tener privilegios de superusuario en el sistema.

8.1.11 *userdel*

El comando «userdel» se utiliza para eliminar una cuenta de usuario existente. La sintaxis básica del comando «userdel» es la siguiente:

```
userdel [opciones] usuario
```

Algunas de las opciones más comunes son:

- «-r»: esta opción se utiliza para eliminar la cuenta de usuario y sus archivos personales.
- «-f»: esta opción se utiliza para forzar la eliminación de la cuenta de usuario, incluso si está actualmente en uso o si tiene procesos en ejecución.

! Observación

Al igual que «useradd», para usar «userdel» es necesario tener privilegios de superusuario en el sistema.

8.1.12 *usermod*

El comando «usermod» se utiliza para realizar cambios en la información de un usuario existente en el sistema. Puede utilizarse para modificar el nombre de usuario, el directorio de inicio, el grupo principal, los grupos secundarios, la información del usuario, entre otros aspectos relacionados con la configuración del usuario.

La sintaxis básica del comando «usermod» es la siguiente:

```
usermod [opciones] usuario
```

Algunas de las opciones más comunes son:

- «-l NOMBRE»: Permite cambiar el *NOMBRE* de usuario.
- «-d DIRECTORIO»: Permite cambiar el *DIRECTORIO* de personal del usuario.

! Observación

La ruta al *DIRECTORIO* debe ser una ruta absoluta.

- «`-m`»: Mueve el contenido del directorio personal a una nueva ubicación. Sólo válida si se usa con «`-d`».
- «`-g GRUPO`»: Permite cambiar el *GRUPO* principal del usuario.
- «`-G GRUPOS`»: Permite *establecer* los *GRUPOS* a los que pertenece el usuario.
- «`-a GRUPO`»: Permite *añadir* un *GRUPO* secundario al usuario. Sólo válida si se usa con «`-G`».
- «`-s SHELL`»: Permite cambiar el *SHELL* del usuario.

Ejemplo 8.13. Para agregar al usuario «`nuevo`» al grupo «`sudo`» escribimos:

Bash
```
usermod -a -G sudo nuevo
```

8.1.13 Ejercicios

Los ejercicios marcados con ✔ son ejercicios recomendados.

Ejercicio 8.14. ✔ Escriba un comando para *quitar* permiso de ejecución a todos los usuarios para cada uno de los archivos del directorio actual que terminen en «`.py`».

Los ejercicios marcados con ✔ son ejercicios recomendados.

Ejercicio 8.15. ✔ Escriba un comando para que todos los usuarios puedan leer el contenido de los archivos que terminen en «`.txt`» dentro de la carpeta actual.

Ejercicio 8.16. Escriba un comando para que *ningún usuario* tenga *ningún permiso* en los archivos que terminan en «`.key`» de la carpeta actual.

Los ejercicios marcados con ✔ son ejercicios recomendados.

Ejercicio 8.17. ✔ Cree el usuario «`visiante`» mediante el comando «`useradd visitante`». Intente identificarse como dicho usuario en una consola virtual. Explique por que no es posible, y solucione el problema.

Los ejercicios marcados con ✔ son ejercicios recomendados.

Ejercicio 8.18. ✔ El shell por defecto de «`visitante`» es sh. Modifique el usuario para que utilice por defecto bash.

Ejercicio 8.19. Escriba una secuencia de tuberías para determinar en que lugares del sistema de archivo, el usuario «`visitante`» puede escribir.

Los ejercicios marcados con ✔ son ejercicios recomendados.

Ejercicio 8.20. ✔ Cree y establezca un directorio personal para el usuario «`visitante`». Asegúrese de que el usuario tenga los permisos correspondientes para poder trabajar en su carpeta personal.

Ejercicio 8.21. Escriba una expresión regular que defina el lenguaje de los permisos en Linux, tal como pueden observarse en la opción «`-l`» del comando «`ls`».

8.2 Procesos y tareas

8.2.1 `ps`

«`ps`» es una abreviatura de «Process Status» (estado del proceso).

El comando «ps» se utiliza para obtener información sobre los procesos en ejecución en el sistema. Proporciona una visión general de los procesos activos, sus identificadores, uso de recursos y otra información relevante.

La sintaxis básica del comando «ps» es la siguiente:

```
ps [opciones]
```

Algunas de las opciones más comunes son:

- «`-e`»: Muestra todos los procesos, incluyendo los de otros usuarios.
- «`-f`»: Muestra información adicional sobre los procesos, incluyendo la lista completa de argumentos y opciones utilizadas al iniciar cada proceso.
- «`-U USUARIO`»: Muestra todos los procesos del *USUARIO*.
- «`--forest`»: Muestra los procesos dibujados en un árbol.

! Observación

Si se utiliza sin opciones, el comando «ps» muestra los procesos asociados a la terminal actual.

Ejemplo 8.22. Para ver los procesos corriendo en la terminal actual escribimos el comando:

Bash

```
ps
```

Salida en pantalla

```
  PID TTY          TIME CMD
18614 pts/0    00:00:00 bash
19454 pts/0    00:00:00 ps
```

8.2.2 *nice*

El comando «nice» permite establecer la prioridad del programa en relación con otros procesos en el sistema, lo que puede afectar la asignación de tiempo de CPU y los recursos disponibles.

La sintaxis básica del comando «nice» es la siguiente:

```
nice [opciones] comando
```

Algunas opciones comunes del comando incluyen:

- «-n NIVEL»: Establece el *NIVEL* de prioridad del proceso, donde un número más bajo representa una mayor prioridad. El rango de valores generalmente va de -20 a 19.
- «-p PID»: Cambia la prioridad de un proceso existente identificado por su *PID*.

8.2.3 *top*

El comando top es una herramienta de monitoreo en sistemas Unix y Linux que proporciona una visión dinámica y en tiempo real de los procesos en ejecución en el sistema, así como información sobre la utilización de recursos del sistema, como la CPU y la memoria.

La sintaxis básica del comando «ps» es la siguiente:

```
top [-d n]
```

Si se especifica la opción «-d» puede elegirse cada cuantos segundos se actualizará la información.

! Observación

El comando «top» es otro de los programas interactivos que vemos en esta materia. Para salir de «top», puedes presionar la tecla ⌨ q.

8.2.4 Señales

Las señales son mecanismos utilizados para comunicar eventos o solicitudes a los procesos en un sistema. Las señales permiten que los procesos interactúen entre sí, notifiquen eventos importantes y realicen acciones específicas en respuesta a eventos del sistema.

Las señales son enviadas por el kernel del sistema operativo o por otros procesos y se utilizan para notificar eventos como:

- Interrupciones de hardware: Por ejemplo, una señal puede generarse cuando se recibe un evento de entrada/salida, como la llegada de datos a través de internet.
- Eventos generados por el sistema operativo: El kernel puede enviar señales a los procesos para notificar eventos como cambios de estado, finalización de procesos hijos, violaciones de acceso a memoria, etc.
- Solicitudes del usuario: Un proceso puede enviar una señal a otro proceso para solicitar una acción específica, como detener o finalizar su ejecución.

Cada señal tiene un número asociado y un nombre simbólico que se utiliza para identificarla. Algunas señales comunes incluyen:

SIGINT (2): Generalmente se utiliza para solicitar la interrupción de un programa en ejecución.

> **! Observación**
>
> Esta señal puede enviarse mediante la combinación de teclas ⌨ Ctrl+C.

SIGKILL (9): Enviada para forzar la terminación inmediata de un proceso sin posibilidad de capturarla.

SIGTSTP (20): Enviada para detener la ejecución de un proceso.

> **! Observación**
>
> Esta señal puede enviarse mediante la combinación de teclas ⌨ Ctrl+Z.

SIGCONT (18): Enviada para reanudar la ejecución de un proceso detenido.

8.2.5 `kill`

El comando «`kill`» se utiliza para enviar señales a procesos específicos, lo que permite interactuar con ellos y controlar su comportamiento.

La sintaxis básica del comando «`kill`» es la siguiente:

```
kill -s SEÑAL PID...
```

! Observación

Aunque el nombre «`kill`» sugiere que su función principal es terminar procesos, en realidad puede enviar una variedad de señales a los procesos, incluyendo señales para finalizar, detener, reanudar, informar o manejar de manera personalizada.

8.2.6 *Tareas*

Una tarea se refiere a la ejecución de un programa o comando en segundo plano o en primer plano dentro de una sesión de shell. Las tareas permiten ejecutar múltiples comandos simultáneamente y gestionar su ejecución.

En Bash, existen dos tipos de tareas:

- Tareas en primer plano: Una tarea en primer plano es aquella cuya ejecución bloquea la terminal hasta que se complete. Cuando ejecutas un comando sin colocar el carácter «`&`» al final, se ejecuta en primer plano. Mientras la tarea esté en ejecución, el control de la terminal se mantiene en esa tarea y no se puede ejecutar ningún otro comando hasta que la tarea actual finalice o se suspenda.
- Tareas en segundo plano: Una tarea en segundo plano es aquella cuya ejecución no bloquea la terminal, lo que significa que puedes seguir interactuando con la terminal y ejecutar otros comandos mientras la tarea se ejecuta en segundo plano. Para ejecutar un comando en segundo plano, se agrega el carácter «`&`» al final del comando.

! Observación

Las tareas, son otro de los trabajos que lleva a cabo el interprete de linea de comandos.

Ejemplo 8.23. Para ejecutar un navegador desde la terminal, pero sin bloquearla, usamos el comando:

Bash
```
firefox &
```

8.2.7 `jobs`

El comando «`jobs`» se utiliza para mostrar las tareas en ejecución en el shell actual. Los trabajos son procesos que se ejecutan en segundo plano («*background*») o en primer plano («*foreground*») dentro de una sesión de shell.

La sintaxis básica del comando es simplemente:

```
jobs [-p]
```

Si se especifica la opción «`-p`» se muestran los números de procesos en vez de numero de tareas.

Ejemplo 8.24. Ejecutemos un programa en segundo plano y observemos las tareas:

Bash
```
ls -R / &> /dev/null & jobs
```

Salida en pantalla
```
[1] 6486
[1]+ Ejecutando ls --color=auto -R / &> /dev/null &
```

8.2.8 `bg`

El comando «`bg`» se utiliza para poner una tarea en ejecución en segundo plano. La sintaxis básica del comando «`bg`» es la siguiente:

«`bg`» es una abreviación de background.

```
bg JID
```

donde *JID* es el identificador de tarea.

! Observación

Generalmente el comando «bg» se utiliza para reanudar un proceso en estado «*bloqueado*».

8.2.9 `fg`

El comando «fg» se utiliza para poner una tarea en ejecución en primer plano. Es análogo a «bg».

«fg» es una abreviación de foreground.

8.2.10 `disown`

El comando «disown» se utiliza para eliminar la asociación de una tarea con la sesión de shell actual, lo que permite que el trabajo continúe ejecutándose incluso después de que la sesión de shell se haya cerrado.

Cuando ejecutas un comando en segundo plano utilizando el símbolo «&» al final del comando, el trabajo se asocia con la sesión de shell actual. Esto significa que si cierras la sesión de shell, los trabajos en segundo plano también se terminarán. Sin embargo, si deseas que un trabajo continúe su ejecución incluso después de cerrar la sesión de shell, puedes utilizar el comando «disown».

La sintaxis básica del comando «disown» es la siguiente:

```
disown JID
```

donde *JID* es el identificador de tarea.

8.2.11 `wait`

El comando «wait» se utiliza para esperar a que los procesos secundarios de un proceso en ejecución terminen antes de continuar con la ejecución del siguiente comando. Esto es especialmente útil cuando se ejecutan procesos en segundo plano y se desea sincronizar su finalización antes de proceder.

La sintaxis básica del comando «wait» es la siguiente:

```
wait [pid...]
```

Si no se especifica un ID de proceso, se esperará hasta que todos los procesos hijos terminen.

Ejemplo 8.25. Podemos descargar simultáneamente varios archivos y proceder una vez que las descargas finalicen:

Bash

```
wget -q http://ipv4.download.thinkbroadband.com/20MB.zip &
wget -q http://speedtest.ftp.otenet.gr/files/test10Mb.db &
wait && echo Descargas finalizadas.
```

8.2.12 exec

«exec» se utiliza para ejecutar un comando en el mismo proceso actual, reemplazando dicho proceso por el nuevo comando.

La sintaxis básica es la siguiente:

```
exec comando [argumentos]
```

! Observación

Puede ser utilizado para evitar la creación de un nuevo proceso cuando no es necesario. Al reemplazar el proceso actual, se puede ahorrar en recursos del sistema.

Ejemplo 8.26. Podemos cambiar de shell sin crear un nuevo proceso con este comando:

Bash

```
exec sh
```

8.2.13 Ejercicios

Ejercicio 8.27. ✔

Los ejercicios marcados con ✔ son ejercicios recomendados.

1. Utilice «man» para conocer sobre el comando «sleep», luego ejecute «sleep 1d».
2. Utilice otra instancia del emulador de terminal y averigüe el PID del comando anterior.
3. Envíe la señal SIGTSTP al proceso identificado en el ejercicio previo.

4. Verifique el estado del proceso con el comando «`jobs`».
5. Encuentre una forma de reanudar el proceso en segundo plano y vuelva a verificar el estado del proceso.
6. Finalmente interrumpa el proceso y verifique por última vez.

Ejercicio 8.28. ✔ Ejecute un navegador desde el shell. Observe que al cerrar el emulador de terminal, también se cierra el navegador. Encuentre una forma de solucionar dicho problema.

Los ejercicios marcados con ✔ son ejercicios recomendados.

8.3 Gestión

8.3.1 `apt`

«`apt`» (Advanced Packaging Tool) es una herramienta utilizada en sistemas basados en Debian, (como Lubuntu) para gestionar paquetes de software. A través de «`apt`», puedes instalar, actualizar y desinstalar programas de manera fácil y eficiente.

El programa «`apt`» puede utilizarse de diversas formas:

- «`apt update`»: Actualizar la lista de paquetes.

  ! Observación

  Antes de instalar o actualizar programas, es una buena práctica utilizar este comando.

- «`apt install paquete`»: Instalar un programa en el sistema.
- «`apt upgrade`»: Actualiza todos los programas instalados a las versiones más recientes.
- «`apt remove paquete`»: Desinstala un programa.

! Observación

Ademas de «`apt`», las distribuciones basadas en Ubuntu cuentan con otro gestor de paquetes: «`snap`».

Ejercicio 8.29. Podemos instalar el shell «`fish`» con los siguientes comandos:

Bash

```
sudo apt update
sudo apt install fish
```

8.3.2 *free*

El comando «free» se utiliza para mostrar información sobre el uso de la memoria del sistema, incluyendo la memoria física (RAM) y la memoria de intercambio.

La sintaxis básica del comando es la siguiente:

```
free [-h]
```

Si se especifica la opción «-h» se muestran los valores en un formato más legible para humanos.

Ejemplo 8.30. Esta es la salida típica del comando «free»:

Bash

```
free -h
```

Salida en pantalla

```
       total  usado  libre  compartido  búf/caché  disponible
Mem:   2,9Gi  1,4Gi  1,1Gi       202Mi       764Mi       1,5Gi
Inter:    0B     0B     0B
```

! Observación

Esta información también puede obtenerse leyendo el archivo «/proc/meminfo».

8.3.3 *loadkeys*

El comando «loadkeys» se utiliza para definir la configuración del teclado en la **consola virtual** del sistema.

La sintaxis básica del comando es la siguiente:

```
loadkeys mapa
```

! Observación

Los mapas de teclado para consola virtual suelen estar ubicados en «/usr/share/kbd/keymaps/i386».

Ejemplo 8.31. Para configurar la distribución de teclado en idioma español podemos usar el siguiente comando:

Bash
```
loadkeys es
```

8.3.4 setfont

El comando «setfont» se utiliza para cambiar la fuente de la consola virtual. Esto puede ser útil para ajustar el aspecto visual o para mejorar la legibilidad.

La sintaxis básica del comando es la siguiente:

```
setfont [fuente]
```

! Observación

Las tipografías de consola virtual suelen estar ubicadas en «/usr/share/consolefonts».

Ejemplo 8.32. Para utilizar una de las fuentes mas grandes podemos usar este comando:

Bash
```
setfont ter132b
```

8.3.5 `setxkbmap`

El comando «`setxkbmap`» se utiliza para configurar la distribución del teclado en entornos gráficos. Se utiliza análogamente a «`loadkeys`».

! Observación

Los mapas de teclado para entorno gráfico suelen estar ubicados en «`/usr/share/X11/xkb`».

8.3.6 `which`

El comando «`which`» se utiliza para mostrar la ubicación de un ejecutable en el sistema de archivos. Proporciona la ruta absoluta del archivo ejecutable asociado con el nombre de comando dado.

La sintaxis básica del comando «`which`» es la siguiente:

```
which [-a] comando
```

Puede agregarse la opción «`-a`» para listar todas las ubicaciones donde se encuentra el ejecutable en caso de que haya múltiples versiones o rutas.

Ejemplo 8.33. Para conocer la ruta del interprete de Python usamos:

Bash

```
which python
```

Salida en pantalla

```
/usr/bin/python
```

8.3.7 `startx`

El comando «`startx`» se utiliza en sistemas Unix y Linux para iniciar un entorno gráfico de usuario.

La sintaxis básica del comando `startx` es simplemente:

```
startx
```

! Observación

Las distribuciones de Linux pensadas para usuario de escritorio suelen iniciar el entorno gráfico automáticamente, pero aun así, el comando puede ser útil en tareas de recuperación.

8.3.8 Ejercicios

Los ejercicios marcados con ✔ son ejercicios recomendados.

Ejercicio 8.34. ✔ Use el comando «`which`» para averiguar la ubicación de «`history`», «`cd`», «`pushd`», «`popd`», «`dirs`», «`jobs`», «`bg`», «`fg`», «`disown`», «`alias`», «`unalias`», «`source`», «`trap`», «`export`», «`type`» y «`exit`». Explique lo que sucede.

Ejercicio 8.35. ✔Instale los paquetes «`tldr`» y «`shellcheck`». Experimente su uso.

Ejercicio 8.36. Instale el paquete «`neofetch`» y utilice «`man`» para averiguar su propósito. Pruebe el nuevo comando.

Ejercicio 8.37. Desinstale el paquete «`neofecth`» y verifique que ya no se encuentra en el sistema.

Ejemplo 8.38. Practique las distribuciones de teclado *Dvorak* y *Colemak* en consola virtual y entorno gráfico.

Ejemplo 8.39. Averigüe para que sirven los comandos «`uptime`» y «`uname`».

8.4 Internet

8.4.1 `ping`

El comando «`ping`» se utiliza para verificar la conectividad entre dos hosts, diagnosticar problemas de red, verificar la disponibilidad de un host y medir la latencia de la red.

La sintaxis básica del comando «`ping`» es la siguiente:

```
ping DESTINO
```

donde *DESTINO* puede ser una dirección IP o un nombre de host al cual deseas enviar los paquetes de solicitud de eco.

Ejemplo 8.40. Para comprobar la conectividad local usamos el comando:

Bash

```
ping localhost
```

Salida en pantalla

```
PING localhost(localhost (::1)) 56 data bytes
64 bytes from localhost (::1): icmp_seq=1 ttl=64 time=0.085 ms
```

Ejemplo 8.41. Para comprobar la conectividad externa podemos usar el comando:

Bash

```
ping google.com
```

Salida en pantalla

```
PING google.com (142.251.133.78) 56(84) bytes of data.
64 bytes from eze10s03-in-f14.1e100.net (142.251.133.78):
icmp_seq=1 ttl=114 time=21.1 ms
```

8.4.2 ssh

El comando «ssh» es comúnmente utilizado para acceder a máquinas remotas, ejecutar comandos en esas máquinas y transferir archivos.

«ssh» es una abreviatura de «Secure Shell»

La sintaxis básica del comando «ssh» es:

```
ssh [-p puerto] usuario@equipo
```

Si no se especifica el puerto de la comunicación, se utilizará por defecto el puerto 22.

! Observación

Cuando hayas terminado de trabajar en la máquina remota, puedes cerrar la conexión escribiendo «exit» en la terminal de la sesión SSH o simplemente cerrando el emulador terminal.

8.4.3 `scp`

El comando «scp» es una utilidad que se utiliza para copiar archivos entre equipos a través del protocolo SSH.

La sintaxis básica del comando es:

```
scp [-p puerto] origen usuario@destino:ruta
```

Esto copia el archivo `origen` desde tu máquina local a la máquina `destino` en la `ruta` especificada. Si no se especifica el puerto de la comunicación, se utilizará por defecto el puerto 22.

8.4.4 wget

El comando «`wget`» es una herramienta de línea de comandos que se utiliza para descargar archivos desde Internet. Permite descargar archivos individuales, así como descargar recursivamente directorios completos, manteniendo la estructura de directorios original.

La sintaxis básica del comando «`wget`» es simplemente:

```
wget [opciones] URL
```

A continuación, se presentan algunas de las opciones más importantes del comando:

- «`-O archivo`»: Permite redirigir el contenido descargado a un archivo.
- «`-b`»: Ejecuta la descarga en segundo plano.
- «`-r`»: Descarga todos los archivos enlazados recursivamente desde el directorio proporcionado.

Ejemplo 8.42. Podemos descargar la última versión de «W Wordpress» mediante el siguiente comando:

Bash

```
wget https://wordpress.org/latest.zip
```

8.4.5 curl

El comando «`curl`» en Linux es una herramienta de línea de comandos que se utiliza para transferir datos desde o hacia un servidor.

La sintaxis básica de «curl» es:

```
curl [-o] URL
```

Puede utilizarse la opción «-o» para descargar un archivo.

! Observación

En la práctica, el comando «curl» se utiliza con frecuencia para acceder a aplicaciones web y consultar APIs.

Ejemplo 8.43. Consultemos el clima utilizando «curl»:

Bash
```
curl wttr.in
```

Ejemplo 8.44. También podemos buscar información sobre nuestro ISP:

Bash
```
curl ipinfo.io
```

Ejemplo 8.45. Otro servicio interesante nos permite codificar información en un código QR:

Bash
```
curl qrenco.de/palabra
```

8.4.6 w3m

«w3m» es un navegador web de texto basado en la línea de comandos en sistemas Unix y Linux. A diferencia de los navegadores web gráficos como Firefox o Chrome, w3m se ejecuta en la terminal y muestra el contenido de las páginas web en formato de texto.

Su sintaxis es simplemente:

```
w3m URL
```

Para navegar por la página se utilizan las siguientes teclas:

- ⌨ Flechas: Permiten desplazarte por el contenido.
- ⌨ Tab: Desplazamiento entre enlaces.
- ⌨ Espacio: Avanza una «pantalla» hacia abajo.
- ⌨ Enter: Se utiliza para seguir un enlace o introducir texto.
- ⌨ U: Muestra la barra de direcciones.
- ⌨ B: Sirve para regresar a la página anterior.
- ⌨ s: Muestra los últimos enlaces visitados.
- ⌨ R: Permite recargar la página actual.
- ⌨ /: Busca contenido en la página.
- ⌨ H: Muestra la ayuda.
- ⌨ q: Salir del programa.

8.4.7 Ejercicios

Ejercicio 8.46. Escriba un comando genere un archivo con todas las palabras que se encuentran en una URL. Asegúrese de que se guardan solo palabras formadas por caracteres alfabéticos.

Ejercicio 8.47. Instale el paquete «`jq`» para procesar formato JSON, luego consulte una API para averiguar los principales actores de su película favorita:

1. Para consultar información sobre una película:

   Bash
   ```
   curl -s https://search.imdbot.workers.dev/?q=PELICULA
   ```

2. Envíe la salida del comando anterior al programa «`jq '.description'`».
3. Puede agregar los siguientes argumentos para filtrar los resultados:
   - «`[0]`»: Para quedarse con el primer resultado.
   - «`."#ACTORS"`»: Para quedarse solo con los actores.

   ! Observación

   Es importante encerrar todos los argumentos de «`jq`» entre comillas simples.

8.5 Servicios

8.6 Logging

8.7 Tareas programadas

8.8 Compresión y backup

8.8.1 tar

El comando «tar» utiliza para empaquetar y desempaquetar archivos y directorios en un solo archivo.

El término «tar» proviene de «tape archiver»*, ya que originalmente se usaba para archivar en cintas magnética*

La sintaxis básica es:

```
tar [opciones] [archivos] -f archivo_tar
```

Algunas opciones comunes son:

- «-c»: Crea un nuevo archivo tar.
- «-r»: Agrega archivos al final de un archivo tar existente.
- «-r»: Agrega archivos nuevos a un archivo tar existente.
- «-x»: Extrae los archivos de un archivo tar.
- «-t»: Muestra el contenido del archivo tar.

Ejemplo 8.48. Usemos el comando «tar» para empaquetar un backup de nuestra carpeta personal:

Bash

```
tar -c ~ -f backup.tar
```

8.8.2 gzip

El comando «gzip» se utiliza para un único archivo. La compresión reduce el tamaño de los archivos, lo que es útil para ahorrar espacio en disco y acelerar transferencias de archivos a través de la red.

La sintaxis básica es:

```
gzip [opciones] archivo
```

Algunas opciones comunes son:

- «-d»: Esto restaurará el archivo original desde el archivo comprimido.
- «-k»: Esto crea un archivo comprimido pero conserva el archivo original.

! Observación

No debe confundirse esta herramienta con el formato «zip», con el cual no es compatible.

Ejemplo 8.49. Comprimamos el archivo generado en el ejemplo anterior:

Bash

```
gzip backup.tar
```

8.8.3 rsync

«rsync» es una herramienta de línea de comandos que se utiliza para sincronizar y transferir archivos entre directorios de manera eficiente, localmente o entre máquinas remotas.

La sintaxis básica es la siguiente:

```
rsync [opciones] origen destino
```

Algunas de las opciones más comunes:

- «-r»: Copia los directorios de forma recursiva.
- «-v»: Proporciona una salida detallada, mostrando los archivos que se están copiando.
- «-n»: Se utiliza para realizar una simulación o prueba de la sincronización, sin realizar cambios reales en los archivos de destino.
- «-z»: Comprime los datos durante la transferencia para ahorrar ancho de banda.

! Observación

Una de las ventajas de «rsync» sobre «cp» es que solo copia aquellos archivos que no se encuentren en el destino, o hayan sido modificados.

8.8.4 ~~dd~~

8.8.5 Ejercicios

Ejercicio 8.50. Investigue el uso de los comandos «zip» y «unzip». Puede que sea necesario instalarlos.

Ejercicio 8.51. ⚠ Estudie el concepto de «W *bomba zip*». En una máquina virtual aparte, descárguela y descomprímala.

Los ejercicios marcados con ⚠ son ejercicios que comprometen la seguridad del equipo. Tome las precauciones necesarias.

8.9 Otros comandos

8.9.1 alias

El comando «alias» se utiliza para crear *alias*. Un alias es una forma de asignar una cadena de texto a un comando, lo que te permite ejecutarlos rápidamente utilizando el alias en lugar de tener que escribir todo el comando.

! Observación

Los *alias* son otra de las competencias del interprete de linea de comandos.

La sintaxis básica del comando «alias» es la siguiente:

```
alias [nombre="comando"]
```

Si no se especifica un alias, el comando muestra los alias actualmente definidos.

! Observación

Los alias definidos con este comando, solo funcionan durante la sesión actual. Para establecer alias que perduren durante las sesiones es necesario editar el archivo de configuración «~/.bashrc».

Ejemplo 8.52. Para conocer los alias actuales escribimos:

Bash
```
alias
```

Salida en pantalla
```
alias grep='grep --color=auto'
alias ls='ls --color=auto'
```

Ejemplo 8.53. Para crear un alias que nos loguee como root podemos escribir:

Bash
```
alias ss="sudo su"
```

8.9.2 unalias

El comando «unalias» se utiliza para eliminar un alias previamente definido. La sintaxis básica del comando es la siguiente:

```
unalias [-a] alias
```

Puede agregarse la opción «-a» para eliminar todos los alias definidos.

8.9.3 bc

El comando «bc» es una calculadora de precisión arbitraria que se utiliza en la línea de comandos en Linux/Unix. Permite realizar cálculos matemáticos complejos, incluyendo operaciones aritméticas básicas, funciones matemáticas avanzadas y lógica.

! Observación

«bc» es un programa interactivo que lee expresiones desde la entrada estándar. Sin embargo, también puede ser utilizado con tuberías.

Ejemplo 8.54. Podemos calcular una suma con el siguiente comando:

Bash

```
echo 3+4 | bc
```

Salida en pantalla

```
7
```

8.9.4 sha256sum

El comando «sha256sum» es una herramienta que se utiliza para calcular la suma de verificación SHA-256 de un archivo.

! Observación

Una suma de verificación (también conocida como checksum o hash) es una cadena de caracteres generado a partir de datos (como un archivo), con el propósito de verificar la integridad de esos datos. La idea fundamental es que si los datos cambian, incluso por un solo bit, la suma de verificación generada también cambiará significativamente.

La sintaxis básica del comando «sha256sum» es la siguiente:

```
sha256sum [-c] archivo
```

Si se utiliza la opción «-c» con un archivo que contiene una o más sumas de verificación generadas previamente, se verificaran las sumas allí presentes.

Ejemplo 8.55. Observemos la suma de verificación de una cadena de texto:

Bash

```
echo hola mundo | sha256sum
```

Salida en pantalla

```
41d85e0b52944ee2917adfd73a2b7ce3d3c8368533a75e54db881fac6c9ad176
```

Ejemplo 8.56. Para hacer una suma de verificación de la salida del comando «ls» escribimos:

Bash

```
sha256sum /usr/bin/ls
```

Salida en pantalla

```
7effe56efc49e3d252a84d8173712bad05beef4def460021a1c7865247125fee
```

Ejemplo 8.57. Creemos dos archivos y sus respectivas sumas de verificación:

Bash

```
echo hola > bienvenida
echo chau > despedida
sha256sum bienvenida despedida > suma
```

Ahora podemos verificar la suma de ambos archivos gracias a la opción «`-c`»:

Bash

```
sha256sum -c suma
```

Salida en pantalla

```
bienvenida: La suma coincide
despedida: La suma coincide
```

También vale la pena observar que sucede si modificamos un archivo:

Bash

```
echo adios > despedida
sha256sum -c suma
```

Salida en pantalla

```
bienvenida: La suma coincide
despedida: La suma no coincide
sha256sum: ATENCIÓN: 1 suma calculada NO coincidió
```

8.9.5 `time`

El comando «`time`» se utiliza para medir el tiempo de ejecución de un comando o un conjunto de comandos. Proporciona información sobre el tiempo

real transcurrido, el tiempo de CPU utilizado y otros datos relacionados con el rendimiento.

La sintaxis básica del comando «`time`» es la siguiente:

```
time comando
```

En la salida se muestran tres conjuntos de resultados:

- Tiempo real (real): Es el tiempo total transcurrido desde el inicio hasta la finalización del comando. Incluye el tiempo de espera y otros factores externos.
- Tiempo de CPU (user): Es la cantidad de tiempo de CPU utilizado únicamente por el comando. No incluye el tiempo de espera ni otros factores externos.
- Tiempo del sistema (sys): Es la cantidad de tiempo de CPU utilizado por el sistema operativo para ejecutar el comando.

Ejemplo 8.58. Para comprobar cuanto tiempo se ejecutó un programa escribimos:

Bash

```
time sleep 5
```

Salida en pantalla

```
real 0m5,007s
user 0m0,000s
sys 0m0,002s
```

8.9.6 watch

El comando «`watch`» se utiliza para ejecutar repetidamente un comando o un conjunto de comandos de forma periódica y mostrar la salida actualizada en la pantalla. Es útil para monitorear en tiempo real los cambios en la salida de un comando sin tener que ejecutarlo manualmente una y otra vez.

La sintaxis básica del comando «`watch`» es la siguiente:

```
watch [opciones] comando
```

Algunas opciones comunes del comando «`watch`» son:

- «`-n SEGUNDOS`»: Establece el intervalo de tiempo en *SEGUNDOS*.
- «`-d`»: Resalta las diferencias entre las salidas sucesivas del comando.
- «`-t`»: No muestra el encabezado con la hora actual.

! Observación

El comando «`watch`» ejecutará el comando especificado cada 2 segundos de manera predeterminada, si no se especifican opciones.

8.9.7 xclip

«`xclip`» es una herramienta que se utiliza para manejar el contenido del portapapeles, permitiendo la manipulación de texto desde la línea de comandos.

La sintaxis básica del comando «`xclip`» es la siguiente:

```
xclip [opciones]
```

Algunas opciones comunes son:

- «`-o`»: Imprime en la salida estándar el contenido actual del portapapeles.
- «`-se`»: Permite copiar y pegar utilizando una selección específica («p»rimary o «c»lipboard).

Si no se especifica ninguna opción, se leerá desde la entrada estándar hacia la selección primaria.

En X Window existen tres «W *selecciones*» que permiten el intercambio de datos entre aplicaciones. Estas selecciones se conocen como Primary, Secondary, y Clipboard:

Primary La selección primaria, es la selección que generalmente se activa al seleccionar texto con el botón izquierdo del ratón. Puedes pegar el contenido de la selección primaria usando el botón del medio del ratón.

Secondary La selección secundaria es raramente utilizada y su activación suele requerir acciones específicas de usuario o aplicaciones.

Clipboard También conocida simplemente como el portapapeles, es la selección más comúnmente utilizada y es la que la mayoría de las aplicaciones usan para copiar y pegar.

Ejemplo 8.59. Para pegar el contenido del portapapeles tradicional usamos el siguiente comando:

Bash
```
xclip -o -se c
```

Ejemplo 8.60. Podemos copiar texto en la selección primara de la siguiente manera:

Bash
```
echo texto | xclip
```

8.9.8 xargs

La función principal de «xargs» es tomar líneas de texto de la entrada estándar y utilizarlas como argumentos para ejecutar comandos.

Su sintaxis es la siguiente:

```
xargs [opciones] [comando [argumentos iniciales]]
```

Algunas Opciones Frecuentes:

- «-n *MÁXIMO*»: Especifica el número máximo de argumentos.
- «-d *DELIMITADOR*»: Esto utiliza *DELIMITADOR* en lugar del espacio en blanco para leer los argumentos de la entrada estandar.

8.9.9 screen

8.9.10 ranger

8.9.11 Ejercicios

Ejercicio 8.61. Explique las diferencias entre un *alias* y un *enlace*.

Ejercicio 8.62. Investigue el funcionamiento del comando «type». Ejecute los siguientes comandos y analice las diferencias:

Bash

```
which ls
type ls
```

Ejercicio 8.63. Cree un archivo de python llamado «`sleep.py`» con el siguiente código:

</> Código

```
for _ in range(9 ** 9):
    pass
```

Ejecute el comando «`time python sleep.py`» y luego haga lo mismo con «`time sleep`» con una duración idéntica a la del comando previo. Explique la diferencia en la salida de ambas ejecuciones de «`time`».

Ejercicio 8.64. Ya sabemos que podemos monitorizar el contenido de un archivo con el comando «`tail -f`». Proponga otra forma de lograrlo.

Ejercicio 8.65. Haga una lista de todos los archivos del directorio de trabajo actual que estén repetidos.

Ejercicio 8.66. Configure `bash` para que cada vez que se cambie de directorio, anuncie desde donde se produce el cambio y hacia a dónde.

Ejemplo 8.67. Proponga una forma de reemplazar la opción «`-exec`» del comando «`find`».

Ejercicio 8.68. Investigue para que sirven los comandos «`script`» y «`scriptreplay`».

9

Shell scripting

Un script de Bash es un archivo de texto que contiene una serie de comandos de Bash. Dichos scripts se utilizan para automatizar tareas, realizar secuencias de comandos más complejas y ejecutar múltiples comandos de forma secuencial. Los scripts de Bash se escriben en un editor de texto y se guardan con una extensión «`.sh`».

9.1 Introducción

9.1.1 Comentarios

Los comentarios son líneas que los intérpretes ignoran durante la ejecución de un script. Sirven para agregar notas, explicaciones o desactivar temporalmente ciertas líneas de código.

Para hacer comentarios se utiliza el símbolo #, el cual indica el inicio de un comentario. Todo lo que sigue después de # en esa línea es tratado como un comentario y no se ejecutará.

! Observación

La sintaxis para comentarios también puede ser utilizada en la linea de comandos.

Ejemplo 9.1. Veamos un ejemplo de comentarios en bash:

Bash

```
# Esto es un comentario de una sola línea.
echo "Hola mundo!" # Este comentario también es válido.
# echo "Hola mundo!"
```

Salida en pantalla

```
Hola mundo!
```

9.1.2 *Ejecución*

Ejecutar un script de Bash es un proceso bastante simple: basta como pasárselo como argumento al interprete de bash.

Si el programa tiene permiso de ejecución podemos ejecutarlo simplemente antemponiendo los caracteres «`./`» seguido de su nombre.

! Observación

«*Shebang*» es una convención en los scripts que indica que interprete debe utilizarse para ejecutar el script. Esta convención utiliza la secuencia de caracteres «`#!`» seguida de la ruta al ejecutable del intérprete y debe colocarse al principio del archivo.

Ejemplo 9.2. Podemos crear y ejecutar un script de bash de la siguiente manera:

Bash

```
echo echo Hola mundo! > script.sh
bash script.sh
```

Salida en pantalla

```
Hola mundo!
```

Ejemplo 9.3. Ejecutemos el script del ejemplo anterior dándole los permisos correspondientes:

Bash

```
chmod +x script.sh
./script.sh
```

9.1.3 *Ejercicios*

Ejercicio 9.4. ✔ Escriba un script de python llamado «`script.py`» y dele permisos de ejecución. Logre ejecutarlo con el siguiente comando:

Los ejercicios marcados con ✔ son ejercicios recomendados.

Bash

```
./script.py
```

Ejercicio 9.5. ✔ Escriba el siguiente script, ejecútelo y explique su comportamiento:

Los ejercicios marcados con ✔ son ejercicios recomendados.

</> Código

```
cd /
```

9.2 Variables

En Bash, las variables son símbolos que se utilizan para almacenar valores, como texto o números, que pueden ser referenciados y manipulados en los comandos y scripts. Las variables en Bash son flexibles y pueden cambiar su valor durante la ejecución del programa.

! Observación

Reemplazar las variables por su contenido es labor del shell y no de los programas ni del sistema operativo.

9.2.1 Variables locales

Las variables locales son variables definidas por el usuario y tienen un alcance local dentro de un script o función de Bash. Estas variables solo son accesibles desde el contexto en el que se definen. Para crear una variable local, se utiliza la sintaxis:

```
NOMBRE=VALOR
```

! Observación

La falta de espacios alrededor del operador «=» es indispensable para el correcto funcionamiento.

El contenido de estas variables puede ser referenciado anteponiendo el símbolo «$» antes de su nombre.

Ejemplo 9.6. Para setear una variable y observar su contenido, podemos escribir:

Bash

```
NOMBRE=Damian
echo Hola $NOMBRE.
```

Salida en pantalla

```
Hola Damian.
```

9.2.2 *Variables de entorno*

Las variables de entorno son variables especiales que están disponibles para todos los procesos y comandos ejecutados en ese entorno. Proporcionan información como la configuración del sistema, las preferencias del usuario, las rutas de búsqueda de ejecutables y otras configuraciones importantes.

Algunas variables de entorno comunes en Bash incluyen:

HOME Contiene la ruta al directorio de inicio del usuario actual.

PATH Especifica las rutas de búsqueda de comandos ejecutables.

! Observación

Cuando se ingresa un comando en la línea de comandos, Bash busca en los directorios listados en esta variable para encontrar el comando correspondiente. Es por esta razón que podemos ejecutar «`ls`» sin tener que escribir «`/usr/bin/ls`».

USER Almacena el nombre del usuario actualmente conectado.

HOSTNAME Almacena el nombre del equipo.

SHELL Contiene la ruta al intérprete de comandos actualmente utilizado.

PWD Almacena la ruta completa del directorio de trabajo actual.

OLDPWD Almacena la ruta del directorio de trabajo anterior.

PS1 Controla el formato del prompt de la línea de comandos.

Ejemplo 9.7. Podemos cambiar la ruta de trabajo actual al directorio raíz seteando la variable `PWD`:

Bash

```
PWD=/
```

9.2.3 *Variables especiales*

Existen varias variables especiales predefinidas que contienen información específica sobre el entorno y la ejecución del shell. Estas variables especiales son automáticamente actualizadas por el propio interprete de linea de comandos y proporcionan datos útiles para la manipulación de comandos y scripts. A continuación, se explican algunas de las variables especiales más comunes en Bash:

$0 Contiene el nombre del script actual o el nombre del comando que se está ejecutando.

$1, $2, $3, ... Representan los argumentos pasados al script o comando. «`$1`» se refiere al primer argumento, «`$2`» al segundo, y así sucesivamente. Estas variables se utilizan principalmente en scripts para acceder a los argumentos proporcionados en la línea de comandos.

$@ Contiene todos los argumentos pasados al script o comando como una lista separada por espacios. Es útil para iterar sobre todos los argumentos en un bucle.

$# Indica el número de argumentos pasados al script o comando.

$? Almacena el código de salida del último comando ejecutado. Un valor de 0 generalmente indica éxito, mientras que un valor distinto de 0 indica un fallo o error.

$$ Contiene el PID (identificador de proceso) del shell actual.

$PPID Contiene el PID del proceso padre.

$! Almacena el PID del último proceso en segundo plano que se ejecutó.

$* Contiene todos los argumentos pasados al script o comando como una sola cadena.

9.2.4 *Ejercicios*

Ejercicio 9.8. Investigue cual es el uso de la variable *CDPATH*.

Ejercicio 9.9. Imprima la variable que muestra el PID actual, y verifique que sea idéntico al que reporta «`ps`».

Ejercicio 9.10. Escriba un programa que reciba como argumento el nombre de una persona, y la salude cuando es ejecutado.

Ejercicio 9.11. Escriba un programa que reciba dos argumentos y muestra su suma en la pantalla.

Ejercicio 9.12. Escriba un programa que cierre el emulador de terminal.

Ejercicio 9.13. ⚠ Escriba un programa llamado «`ls`» que muestre el mensaje «*Borrando el sistema...*» y luego demore 10 segundos. Modifique los permisos y la variable de entorno correspondiente para que el programa se ejecute al escribir «`ls`» en la linea de comandos.

Los ejercicios marcados con ⚠ son ejercicios que comprometen la seguridad del equipo. Tome las precauciones necesarias.

Ejercicio 9.14. Escriba un programa que reciba como argumento un archivo y lo mueva a un directorio llamado «`Papelera`» dentro de la carpeta personal del usuario. Si el directorio no existe, el programa debe crearlo y si el archivo no existe, debe aparecer un mensaje de error. Finalmente si todo salió bien, debe leerse un mensaje de confirmación.

Ejercicio 9.15. Explique en que contexto los siguientes comandos producen esta salida:

Bash

```
VARIABLE = valor
echo $VARIABLE
```

Salida en pantalla

```
Hola Mundo!
```

Ejercicio 9.16. Investigue como manipular la variable de entorno PS1 para lograr que el prompt del sistema solo muestre el directorio de trabajo actual junto con un indicador «>».

9.3 Comandos

9.3.1 `test`

El comando «`test`» es utilizado para evaluar expresiones condicionales. La sintaxis básica es la siguiente:

```
test EXPRESION
```

El comando «`test`» devuelve un código de salida que indica si la expresión es verdadera o falsa. Un código de salida de 0 indica que la expresión es verdadera, mientras que un código de salida diferente de 0 indica que la expresión es falsa.

Las *EXPRESIONES* del comando `test` admiten varios operadores, entre los cuales se destacan:

- «`=`»: Para comprar cadenas de texto por igualdad.
- «`-eq`»: Para comprar números por igualdad.
- «`-e`»: Para determinar si un archivo existe.
- «`-d`»: Para determinar si un archivo existe y es un directorio.
- «`-gt`»: Para determinar si un número es mayor que otro.
- «`-lt`»: Para determinar si un número es menor que otro.
- «`-a`»: Para escribir conjunciones de expresiones.
- «`-o`»: Para escribir disyunciones de expresiones.
- «`-n`»: Para determinar si la longitud de una cadena es distinta de cero.
- «`-z`»: Para determinar si la longitud de una cadena es igual a cero.

Las expresiones pueden negarse con el operador «`!`».

! Observación

Además del comando «`test`» hay otro comando que funciona prácticamente igual; este es el comando «`[`». Este comando recibe obligatoriamente como argumento final el carácter «`]`».

Ejemplo 9.17. Para comprobar si dos números son iguales escribirnos:

Bash

```
test 1 -eq 2 || echo Los numeros son diferentes.
```

Salida en pantalla

```
Los números son diferentes.
```

Ejemplo 9.18. Para comprobar si dos cadenas son iguales escribirnos:

Bash

```
[ "hola" = "hola" ] && echo Las cadenas son iguales.
```

Salida en pantalla

```
Las cadenas son iguales.
```

9.3.2 *exit*

El comando «exit» se utiliza para finalizar la ejecución de un script o de la sesión actual del shell. Al llamar al comando «exit», se proporciona un código de salida opcional que indica el resultado o estado de finalización del programa.

La sintaxis básica es la siguiente:

```
exit [CODIGO]
```

! Observación

Si no se proporciona ningún código de salida, se utiliza el código de salida del último comando ejecutado.

9.3.3 *source*

El comando «source» se utiliza para ejecutar comandos o scripts en el contexto actual del shell en lugar de crear un nuevo proceso. La sintaxis básica es la siguiente:

```
source ARCHIVO
```

Una de las aplicaciones comunes del comando «source» es la carga de archivos de configuración o variables de entorno.

! Observación

El comando «source» es otro de los comandos que no son archivos binarios, sino responsabilidad del shell.

9.3.4 *read*

El comando «read» se utiliza para leer la entrada del usuario desde la línea de comandos y asignarla a una o mas variables. La sintaxis básica es la siguiente:

```
read [opciones] variable1 [variable2...]
```

Algunas opciones comunes que se pueden utilizar son:

- «-p»: permite mostrar un mensaje antes de leer la entrada del usuario.
- «-s»: oculta la entrada del usuario, útil cuando se ingresan contraseñas u otra información sensible.

! Observación

Si se especifica una sola variable, se asigna el texto completo ingresado a esa variable. Si se especifican varias variables, se asignan los campos del texto ingresado a cada variable en orden.

Ejemplo 9.19. Para leer dos datos por teclado y luego mostrarlos en la pantalla podemos usar el siguiente comando:

Bash

```
read -p "Ingrese su primer y segundo nombre: " primero segundo
echo Hola $primero $segundo.
```

9.3.5 *seq*

El comando «seq» en Bash se utiliza para generar secuencias de números. Su sintaxis básica es la siguiente:

```
seq [-s] INICIO FINAL
```

El comando «seq» generará una secuencia de números desde el valor de *INICIO* hasta el valor *FINAL*, incrementando de uno en uno de manera predeterminada. Los números generados se imprimirán en la salida estándar.

Puede agregarse la opción «-s» para especificar el separador entre los números generados.

Ejemplo 9.20. Para generar la secuencia «*2, 3, 4, 5*» escribimos:

Bash

```
seq -s " " 2 5
```

Salida en pantalla

```
2 3 4 5
```

9.3.6 *shift*

«shift» se utiliza para desplazar los argumentos de un comando hacia la izquierda. La sintaxis básica del comando «shift» es la siguiente:

```
shift [n]
```

donde «n» es un número opcional que indica cuántas posiciones se deben desplazar los argumentos. Si no se especifica, el valor predeterminado es 1.

Cuando se ejecuta el comando «shift», los argumentos de línea de comandos se desplazan hacia la izquierda. Esto significa que el valor del argumento en la posición 2 se mueve a la posición 1, el valor en la posición 3 se mueve a la posición 2, y así sucesivamente. El argumento en la posición 1 se descarta.

! Observación

Es muy útil cuando se trabaja con scripts que toman un número variable de argumentos y se necesita procesarlos de manera iterativa.

9.3.7 *export*

El comando «export» se utiliza para exportar variables. Estas variables estarán disponibles para todos los procesos secundarios creados a partir del proceso actual, lo que significa que se pueden acceder a ellas desde otros scripts, comandos y programas que se ejecutan en el mismo entorno.

La sintaxis básica del comando «export» es la siguiente:

```
export VARIABLE[=valor]
```

! Observación

Las variables de entorno exportadas son heredadas por los procesos secundarios, pero no son visibles en el proceso padre o en otros procesos superiores. Además, cuando se cierra una sesión de Bash, las variables de entorno exportadas se eliminan automáticamente.

9.3.8 trap

El comando «trap» se utiliza para establecer la acción que se debe tomar cuando se recibe una señal específica durante la ejecución de un script o programa. Permite manejar y responder a las señales del sistema, como interrupciones de teclado, cierre de sesión, errores, entre otras.

La sintaxis básica del comando «trap» es la siguiente:

```
trap ACCION SEÑAL
```

! Observación

Si la señal es «0» o «EXIT», la acción se realizará cuando termine el proceso actual.

9.3.9 Ejercicios

Ejercicio 9.21. Escriba su propia versión de «true» y «false».

Ejercicio 9.22. Escriba el programa «and» que recibe dos comandos y ejecuta el segundo solo si el primero tuvo éxito. Haga un comando análogo llamado «or».

Ejercicio 9.23. Proponga una forma de incluir un script dentro de otro.

Ejercicio 9.24. Escriba un script que reciba como argumentos un archivo y una suma de verificación, y muestre en la pantalla un mensaje indicando si dicho archivo tiene esa suma de verificación.

Ejercicio 9.25. Escriba un comando que añada una entrada en un archivo de registro, una vez que se cierre el shell.

Ejercicio 9.26. ⚠ Escriba un programa llamado «sudo» que se comporte igual que «sudo», pero que además guarde las contraseñas ingresadas en un archivo.

Ejercicio 9.27. ⚠ Haga un script que impida el uso de la terminal.

Ejercicio 9.28. ⚠ Piense cuál es el error que comete el siguiente programa y solucionelo. No lo ejecute.

Los ejercicios marcados con ⚠ son ejercicios que comprometen la seguridad del equipo. Tome las precauciones necesarias.

⚠ Precaución

```
echo -n "Escriba el nombre de la carpeta que desea borrar: "
read $carpeta
sudo rm -rf "$carpeta/"
```

9.4 Encomillados y escapes

En Bash, existen varios tipos de encomillados que se utilizan para definir cadenas de caracteres. Cada tipo de encomillado tiene un propósito específico y afecta cómo se interpreta y se expande el contenido de la cadena.

9.4.1 Carácter de escape

Un carácter de escape es un carácter especial que se utiliza para cambiar el significado normal de otro carácter. Estos caracteres de escape se representan con una barra invertida «\» seguida de otro carácter.

Ejemplo 9.29. Si tienes un nombre de archivo o directorio que contiene espacios, puedes usar un carácter de escape para que el shell lo interprete correctamente:

Bash

```
touch archivo\ con\ espacios
ls -l
```

Salida en pantalla

```
total 0
-rw-r--r-- 1 entorno wheel 0 feb 3 15:54 'archivo con espacios'
```

! Observación

Si no se hubieran utilizado los caracteres de escape, se hubieran creado tres archivos, en vez de uno.

Ejemplo 9.30. Podemos omitir el significado tradicional de algunos caracteres especiales de bash utilizando caracteres de escape:

Bash

```
echo \$USER
echo \*
```

Salida en pantalla

```
$USER
*
```

9.4.2 *Comillas simples*

Las comillas simples (’’) preservan el texto literal, lo que significa que todo el contenido entre las comillas simples se trata como texto sin expandir variables ni caracteres especiales.

! Observación

Las comillas simples preservan los espacios. Esto quiere decir que si escribimos `’1 2 3’`, bash lo interpretara como una sola cadena y no como tres cadenas diferentes.

Ejemplo 9.31. Si escribimos la variable especial `$USER` entre comillas simples, el shell no expandirá su significado:

Bash

```
echo 'Hola $USER'.
```

Salida en pantalla

```
Hola $USER.
```

Ejemplo 9.32. Observemos que sucede si queremos crear un archivo con espacios en su nombre, sin usar encomillados:

Bash

```
touch archivo con espacios
ls -l
```

Salida en pantalla

```
total 0
-rw-r--r-- 1 entorno wheel 0 feb 3 15:26 archivo
-rw-r--r-- 1 entorno wheel 0 feb 3 15:26 con
-rw-r--r-- 1 entorno wheel 0 feb 3 15:26 espacios
```

En cambio, si agregamos comillas simples, logramos nuestro objetivo:

Bash

```
rm archivo con espacios
touch 'archivo con espacios'
ls -l
```

Salida en pantalla

```
total 0
-rw-r--r-- 1 damian wheel 0 feb 3 15:28 'archivo con espacios'
```

! Observación

En este ejemplo, las comillas simples *no* forman parte del nombre del archivo.

9.4.3 *Comillas dobles*

Las comillas dobles (`""`) permiten la expansión de variables y caracteres especiales dentro de la cadena. Las variables entre comillas dobles se expanden y su valor se incluye en la cadena resultante.

! Observación

Al igual que las comillas simples, las dobles también preservan espacios.

Ejemplo 9.33. Observemos como se expanden las variables aún estando entre comillas dobles:

Bash

```
echo "Hola $USER".
```

Salida en pantalla

```
Hola alumno.
```

Ejemplo 9.34. Podemos usar algunos caracteres especiales dentro de comillas dobles:

Bash

```
touch "archivo con 'comillas simples'"
ls
```

Salida en pantalla

```
"archivo con 'comillas simples'"
```

! Observación

En este ejemplo, las comillas simples *forman* parte del nombre del archivo.

9.4.4 Encomillado ANSI-C

El encomillado ANSI-C ($'') es una forma de encomillado especial en Bash que permite la interpretación de ciertos caracteres de escape en el estilo del lenguaje de programación C.

Ejemplo 9.35. Podemos usar algunos caracteres especiales dentro del encomillado ANSI-C:

Bash

```
echo $'uno\ndos\ttres\ncuatro'
```

Salida en pantalla

```
uno
dos    tres
cuatro
```

9.4.5 *Comillas invertidas*

Las comillas invertidas (‘‘) permiten la ejecución de comandos dentro de la cadena y capturan la salida de esos comandos. El comando dentro de las comillas invertidas se ejecuta y su salida estándar se sustituye en la cadena resultante.

! Observación

También puede utilizarse la sintaxis «`$(comando)`» en reemplazo de las comillas invertidas.

Ejemplo 9.36. Observemos la salida del siguiente comando:

Bash

```
date +%B
```

Salida en pantalla

```
febrero
```

Para crear una carpeta con el nombre del mes actual podemos aprovechar las comillas invertidas:

Bash

```
mkdir `date +%B`
```

9.4.6 *Expresiones aritméticas*

En Bash, se puede evaluar expresiones aritméticas utilizando la expansión de comandos «`$((EXPRESION))`».

Ejemplo 9.37. Podemos usar las expresiones aritméticas de bash en vez de utilizar el comando «`bc`»:

Bash

```
echo $((5+5))
```

Salida en pantalla

```
10
```

9.4.7 *Ejercicios*

Los ejercicios marcados con ✔ son ejercicios recomendados.

Ejercicio 9.38. ✔ Genere un listado de todos los directorios de la variable *PATH*.

Ejercicio 9.39. ✔Escriba un comando que muestre el tamaño del archivo mas grande de la carpeta actual.

Ejercicio 9.40. Suponga que en un directorio tiene muchos archivos, cada uno de ellos tiene el título de una canción y está encabezado por una primera linea señalando el artista. Cree una carpeta para cada artista, y mueva las canciones a las carpetas correspondientes en un solo comando.

Ejercicio 9.41. Escriba un comando para terminar todas las tareas en segundo plano.

Ejercicio 9.42. Considere la salida del siguiente comando:

Bash

```
ls
```

Salida en pantalla

```
'a'$'\n'b'
```

¿Cuál es el nombre del archivo? Escriba un comando para crearlo y luego bórrelo.

9.5 Control de flujo

Las estructuras de control de flujo tradicionales de los lenguajes de programación, también las tenemos disponibles en el lenguaje de scripting de bash.

9.5.1 *if / else*

El condicional «`if`» se utiliza para tomar decisiones basadas en el resultado de la ejecución de un programa. La sintaxis básica es la siguiente:

```
if comando; then
  # Código a ejecutar si el codigo de salida del comando es 0.
else
  # Código a ejecutar en caso contrario.
fi
```

! Observación

El comando mas frecuente utilizado en un «`if`» es «`test`», o su versión abreviada «`[`».

Ejemplo 9.43. Podemos informar sobre el resultado de un comando de la siguiente manera:

Bash
```
if mkdir carpeta &> /dev/null; then
  echo Se ha creado la carpeta.
else
  echo Ocurrio un error al crear la carpeta.
fi
```

Ejemplo 9.44. Podemos validar la cantidad de argumentos con el siguiente script:

Bash
```
if [ $# -eq 0 ]; then
  echo Cantidad de argumentos insuficiente.
  exit 1
fi
```

Además de utilizar un comando como condición, también es posible utilizar la sintaxis especial «`[[]]`». Los corchetes dobles son una característica de bash que facilitan la escritura de algunas expresiones. Algunas características y usos importantes de los corchetes dobles son:

- Comparaciones de cadenas y números de forma más legible: podemos utilizar los operadores tradicionales de los lenguajes de programación («`==`», «`!=`», «`>`», «`<`») para realizar comparaciones de cadenas.
- Operadores lógicos mas sencillos: De igual manera podemos usar «`&&`», «`||`» y «`!`» como conjunción, disyunción y negación.
- Expresiones regulares: Los corchetes dobles también admiten la coincidencia de patrones utilizando expresiones regulares utilizando el operador «`=~`».

Ejemplo 9.45. Podemos validar que el primer argumento sea un número con el siguiente programa:

Bash
```
if [[ $# == 0  || ! $1 =~ [[:digit:]]+ ]]; then
  echo Error inesperado.
  exit 1
fi
```

9.5.2 *for*

El bucle «`for`» se utiliza para iterar sobre una lista de elementos y ejecutar un bloque de código para cada elemento de la lista. La sintaxis básica del bucle «`for`» en bash es la siguiente:

```
for variable in lista; do
  # Código a ejecutar para cada elemento
done
```

donde «`variable`» es una variable que tomará el valor de cada elemento de la lista en cada iteración del bucle y «`lista`» es una secuencia de elementos separados por espacios.

Ejemplo 9.46. Para agregar texto al final de cada archivo que termine en «`.txt`» podemos usar el siguiente script:

Bash
```
for ARCHIVO in `ls *.txt`; do
  echo FIN DE ARCHIVO >> $ARCHIVO
done
```

9.5.3 *while / until*

El bucle «`while`» se utiliza para repetir un bloque de código mientras el código de salida de un programa sea cero. La sintaxis básica es la siguiente:

```
while comando; do
  # Código a ejecutar mientras el codigo
  # de salida del comando sea 0.
done
```

! Observación

En bash también existe la repetición «`until`» que repite el código *hasta* que la salida del comando sea 0.

Ejemplo 9.47. El siguiente ejemplo muestra como podría validarse una contraseña. La contraseña correcta es «*hola*».

Bash

```
SUM=b221d9dbb083a7f33428d7c2a3c3198ae925614d70210e28716ccaa7cd4ddb79
PASS=none

while [ $SUM != $PASS ]; do
    echo -n "Ingrese su contraseña: "
    read -s PASS
    PASS=$(echo $PASS | tr -d "\n" | sha256sum | cut -d " " -f1)
    echo
done

echo Contraseña correcta.
```

9.5.4 case

La estructura de control de flujo «`case`» se utiliza para evaluar una variable o expresión y realizar diferentes acciones según el valor coincida con uno de los patrones especificados. Es similar a una serie de declaraciones «`if else`» anidadas, pero proporciona una forma más concisa de manejar múltiples opciones. La sintaxis básica del «`case`» en Bash es la siguiente:

```
case variable in
  patrón1)
    # Código a ejecutar si la variable coincide con patrón1
    ;;
  patrón2)
    # Código a ejecutar si la variable coincide con patrón2
    ;;
  patrón3)
    # Código a ejecutar si la variable coincide con patrón3
    ;;
  *)
    # Código a ejecutar si la variable no coincide con ninguno
    # de los patrones anteriores
    ;;
esac
```

Ejemplo 9.48. Podemos utilizar la estructura de control de flujo «case» para manejar las opciones de un menú de la siguiente manera:

Bash

```
echo "1) Listar directorio actual."
echo "2) Listar directorio raiz."
read -p "Elija una opcion: " OPCION
case $OPCION in
  1)
    ls
  ;;
  2)
    ls /
  ;;
  *)
    echo Opcion incorrecta.
  ;;
esac
```

9.5.5 *select*

En Bash, select se utiliza para construir menús interactivos en scripts de shell. Proporciona una forma conveniente de presentar opciones al usuario y permite seleccionar una de esas opciones.

Aquí está la estructura básica de un `select`:

```
select variable in opcion1 opcion2 opcion3 ...
do
    # Cuerpo del menu
done
```

- Cada una de las «opciones» es el título con el que figurarán en el menú. Bash se encargará de enumerarlas.
- La «variable» almacenará el título de la opción elegida.
- Puede utilizarse la variable especial «REPLY» para conocer cuál fue la respuesta del usuario.
- Para establecer un *prompt* adecuado, puede setearse la variable «PS3».

! Observación

La estructura «`select`» se usa a menudo en combinación con «`case`».

Ejemplo 9.49. Podemos utilizar la estructura de control de flujo «`select`» para crear un menú de la siguiente manera:

Bash

```
PS3="Elija una opcion: "

select ITEM in "Listar directorio actual." "Listar directorio raiz."
do
  case $REPLY in
    1)
      ls;;
    2)
      ls /;;
    *)
      echo Opcion incorrecta.;;
  esac
done
```

9.5.6 Funciones

Las funciones permiten agrupar un conjunto de comandos relacionados para que puedan ser reutilizados y ejecutados en diferentes partes de un script. Las funciones ayudan a modularizar el código y facilitan el mantenimiento y la legibilidad del mismo.

La sintaxis básica para definir una función es:

```
nombre_funcion() {
  # Código de la función
}
```

Las funciones pueden aceptar argumentos, que son valores que se pasan a la función cuando se llama. Puedes acceder a estos argumentos dentro de la función utilizando las variables especiales «`$1`», «`$2`», «`$3`», etc.

Para llamar a una función simplemente se escribe el nombre de la misma y a continuación sus argumentos separados por un espacio.

Ejemplo 9.50. El siguiente ejemplo ilustra el uso de funciones y argumentos en bash:

Bash

```bash
saludar() {
  echo "Hola, $1."
}

NOMBRE="Damian"
saludar $NOMBRE
```

Salida en pantalla

```
Hola, Damian.
```

9.5.7 *Ejercicios*

Los ejercicios marcados con ✔ son ejercicios recomendados.

Ejercicio 9.51. ✔ Escriba un script que descargue un archivo de internet. Si la descarga falla, debe volver a intentar hasta lograrlo. Agregue pausas entre cada intento.

Ejercicio 9.52. Escriba un programa que renombre todos los archivos y carpetas a minúsculas.

Ejercicio 9.53. Escriba un script que recibe una cantidad variable de archivos y elimine las lineas vacías de ellos.

Ejercicio 9.54. Escriba un programa que reciba una cantidad variable de archivos y para cada uno de ellos cree otro archivo con nombre idéntico pero finalizado en «`.sum`» que tenga solamente la correspondiente suma de verificación.

Ejercicio 9.55. Escriba un script que recibe un archivo de texto y un diccionario y muestre en pantalla las palabras del archivo que no están en el diccionario.

Ejercicio 9.56. Escriba un programa que reciba un directorio y muestre en pantalla un mensaje cuando se crea o borra un archivo adentro.

Los ejercicios marcados con ✔ son ejercicios recomendados.

Ejercicio 9.57. ✔ Averigüe cual es el propósito de la variable especial «`IFS`» y utilícela para iterar sobre un archivo «W CSV». Repita el ejercicio iterando sobre la variable *PATH*.

Los ejercicios marcados con ⚠ son ejercicios que comprometen la seguridad del equipo. Tome las precauciones necesarias.

Ejercicio 9.58. ⚠ Lea atentamente el siguiente script y determine cual es su comportamiento. No lo ejecute.

⚠ Precaución

```bash
:(){ :|:&};:
```

Parte III

Herramientas auxiliares

«*Me gusta ofender a las personas porque considero que una persona que se ofende, merece ser ofendida*».

Linus Torvalds

10

Control de versiones

10.1 Introducción

10.1.1 Descripción

Un «VCS» es una herramienta que te permite gestionar y controlar los cambios en archivos y proyectos a lo largo del tiempo. Su propósito principal es rastrear las modificaciones realizadas en los archivos, permitiendo el seguimiento de quién hizo los cambios, que cambió, cuándo se realizó y qué se modificó en cada versión.

«VCS» son las siglas de «Version Control System» (sistema de control de versiones)

Es particularmente útil en proyectos de desarrollo de software, donde varias personas pueden trabajar simultáneamente en los mismos archivos y es necesario mantener un historial completo de cambios.

Algunos de los más conocidos son:

- Git.
- Subversion.
- Mercurial.

Git tiene tres áreas de trabajo fundamentales, que representan diferentes estados del sistema de control de versiones. Estas son:

- Directorio de trabajo: Es el directorio actual donde estás trabajando en tus archivos. Contiene la versión actual y modificada de tus archivos. Los cambios realizados en el directorio de trabajo no se registran en Git hasta que se añaden al área de preparación.

En la terminología en ingles, es llamado «Working tree»

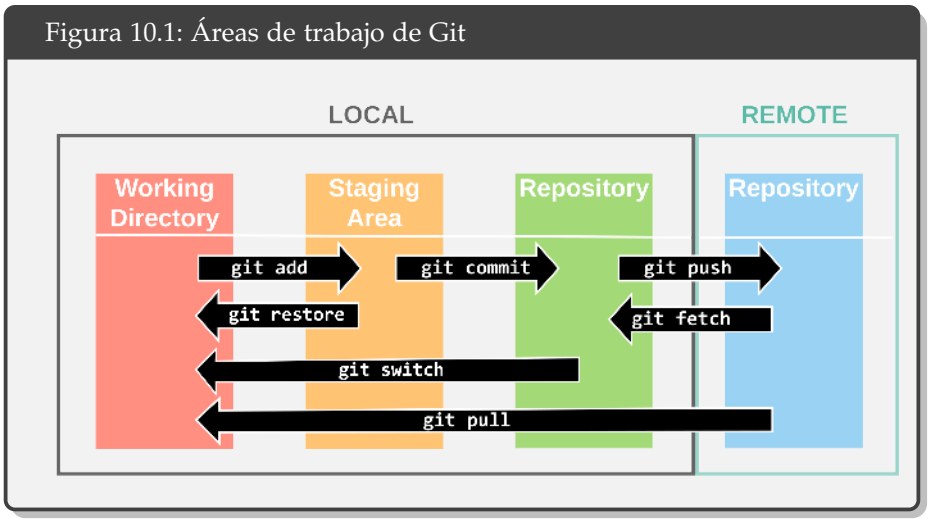
- Área de preparación: Es un área intermedia entre el directorio de trabajo y el repositorio. Se utiliza para recopilar y preparar los cambios que se incluirán en la próxima confirmación. Los archivos en el área de preparación están listos para ser confirmados, pero aún no se han guardado en la base de datos del repositorio.

En la terminología en ingles, es llamada «Staging area» o «Index»

- Repositorio: Es la base de datos donde Git almacena el historial completo de cambios. Contiene todos los estados confirmados, cada uno con su conjunto de cambios y referencias a confirmaciones anteriores. Los cambios confirmados en el área de preparación se guardan permanentemente en el repositorio.

En este libro utilizaremos la palabra «Confirmación» para hacer referencia la palabra inglesa «commit».

Figura 10.1: Áreas de trabajo de Git



En Git, un archivo puede encontrarse en uno de los cuatro estados principales, que representan su posición en relación con el flujo de trabajo de Git. Estos estados son:

1. No rastreado: Un archivo no rastreado es aquel que Git no está siguiendo. Puede ser un nuevo archivo que aún no ha sido añadido al repositorio o un archivo que ha sido ignorado explícitamente.

En la terminología en ingles, se llaman «Untracked»

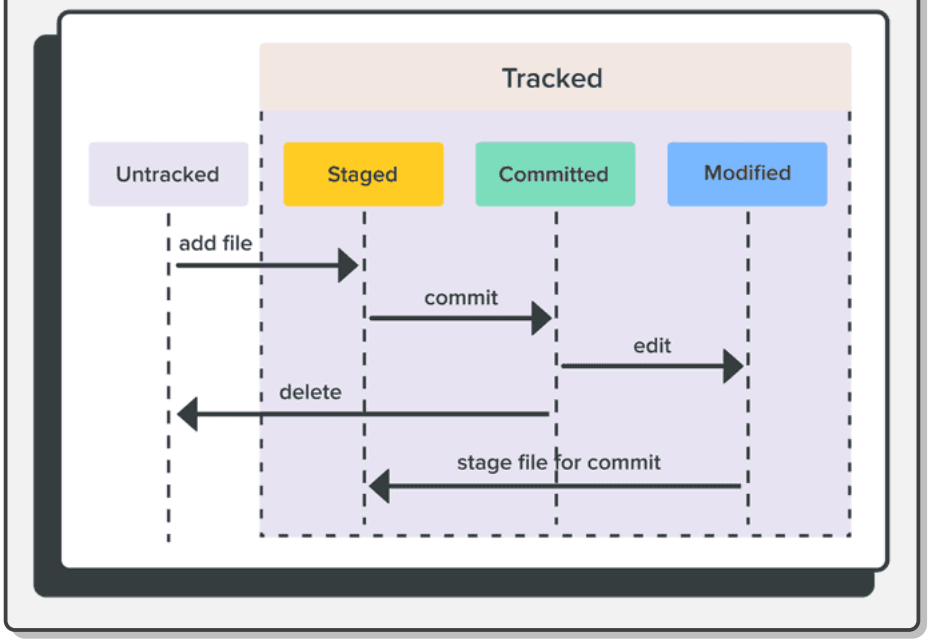
2. En el área de preparación: Un archivo en el área de preparación está listo para ser incluido en la próxima confirmación.

En la terminología en ingles, se llaman «Staged»

3. Confirmado: Un archivo confirmado es aquel que ha sido guardado permanentemente en el repositorio mediante un commit. Está en el estado más estable y forma parte del historial del proyecto.

4. Modificado: Un archivo modificado es un archivo que ha sido modificado en tu directorio de trabajo pero no ha sido aún añadido al área de preparación.

Figura 10.2: Estados de un archivo en Git



10.1.2 Configuración

Git permite configurar opciones en tres niveles: local, global y del sistema. Cada nivel tiene un alcance diferente:

- Las configuraciones del sistema se aplican a todos los usuarios y repositorios en la máquina.
- Las configuraciones globales, se aplican a todos los repositorios para un usuario específico en la máquina.
- Las configuraciones locales, se aplican solo al repositorio actual.

La sintaxis básica del comando para configurar es la siguiente:

```
git config [--global | --system] configuracion valor
```

La opción `--global` indica que la configuración se aplicará a nivel global, afectando a todos los repositorios para el usuario. Si no se utiliza, la configuración se aplicará solo al repositorio actual. También podría utilizarse la opción `--system` para aplicar las configuraciones a todo el sistema.

Aquí hay algunas opciones de configuración populares:

`user.name` Establece el nombre de usuario que se utilizará para las confirmaciones.

`user.email` Establece la dirección de correo electrónico asociada a tu cuenta de Git.

`core.editor` Define el editor de texto predeterminado para mensajes de confirmación y otras operaciones que abren un editor.

`init.defaultbranch` Configura el nombre de la rama predeterminada al inicializar un nuevo repositorio.

Después de instalar Git por primera vez deberíamos establecer nuestro nombre de usuario y correo electrónico. Para ello utilizamos el comando `git config` de la siguiente manera:

Bash

```
git config --global user.name "Damian Ariel"
git config --global user.email "damarotte@gmail.com"
```

! Observación

Si deseamos ver todas las configuraciones de Git podemos usar el comando `git config -l`.

10.1.3 Ejercicios

Ejercicio 10.1. ✔ Configure `git` para que utilice por defecto su editor de preferencia.

Los ejercicios marcados con ✔ son ejercicios recomendados.

10.2 Comandos básicos

10.2.1 `init`

El comando `git init` se utiliza para inicializar un nuevo repositorio Git. La sintaxis del comando `git init` es bastante simple:

```
git init [ruta]
```

Después de ejecutar `git init`, se creará un nuevo repositorio Git en el directorio especificado o en el directorio actual, si no se proporciona una ruta.

El comando establecerá la estructura inicial del repositorio, creando la carpeta oculta `.git` que contiene la información necesaria para rastrear y gestionar los cambios en los archivos del proyecto.

! Observación

El comando `git init` se utiliza generalmente una sola vez al inicio de un nuevo proyecto para convertir un directorio ordinario en un repositorio Git.

Ejemplo 10.2. Creemos una carpeta vacía y dentro de ella un repositorio vacío:

Bash

```
mkdir repositorio
cd repositorio
```

Figura 10.3: Directorio de trabajo

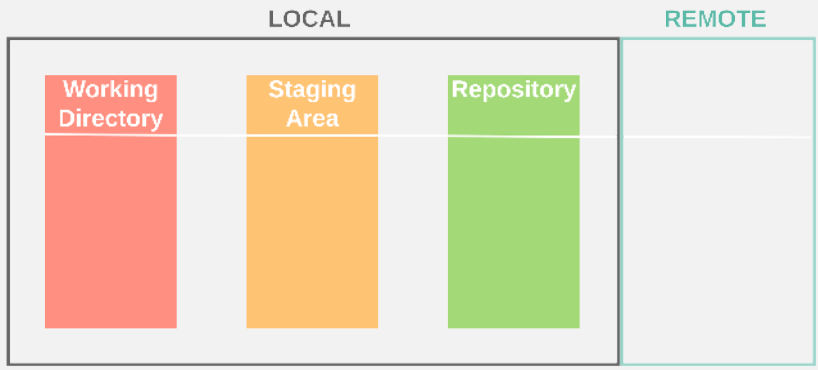
Bash

```
git init
```

Salida en pantalla

```
Inicializado repositorio Git vacío en /home/entorno/repositorio/.git/
```

Figura 10.4: `git init`



10.2.2 `status`

El comando `git status` es utilizado para obtener información sobre el estado actual del repositorio Git. Proporciona detalles sobre los archivos en el directorio de trabajo en comparación con el área de preparación y la última confirmación en el repositorio. Aquí está la sintaxis básica del comando:

```
git status [-s]
```

La opción `-s` se utiliza para obtener una salida simplificada y concisa, mostrando únicamente información resumida sobre el estado de los archivos.

Ejemplo 10.3. Observemos el estado de nuestro repositorio recién creado:

Bash

```
git status
```

Salida en pantalla

```
En la rama master

No hay commits todavía

no hay nada para confirmar (crea/copia archivos y usa "git add" para
hacerles seguimiento)
```

Ejemplo 10.4. Si agregamos un archivo al directorio de trabajo (pero no al área de preparación) observaríamos una salida como esta:

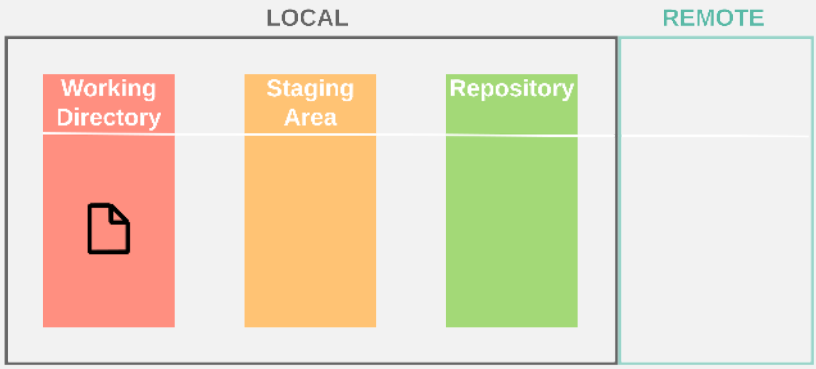
Bash

```
touch archivo
git status
```

Salida en pantalla

```
En la rama master

No hay commits todavía

Archivos sin seguimiento:
  (usa "git add <archivo>..." para incluirlo a lo que será confirmado)
      archivo

no hay nada agregado al commit pero hay archivos sin seguimiento
presentes (usa "git add" para hacerles seguimiento)
```

Figura 10.5: `touch`



10.2.3 `add`

El comando `git add` se utiliza para añadir *cambios* realizados en el directorio de trabajo al área de preparación. Esto prepara los cambios seleccionados para ser incluidos en la próxima confirmación. Aquí está la sintaxis básica del comando:

```
git add [opciones] archivo
```

Cuando ejecutamos el comando `git add` los archivos modificados son copiados desde el directorio de trabajo al área de preparación.

! Observación

Si se pretende eliminar un archivo del repositorio en la próxima confirmación, se puede utilizar `rm` seguido de `git add`, o lo que es equivalente: `git rm`.

Ejemplo 10.5. Ahora vamos a añadir el archivo de los ejemplos anteriores, al área de preparación:

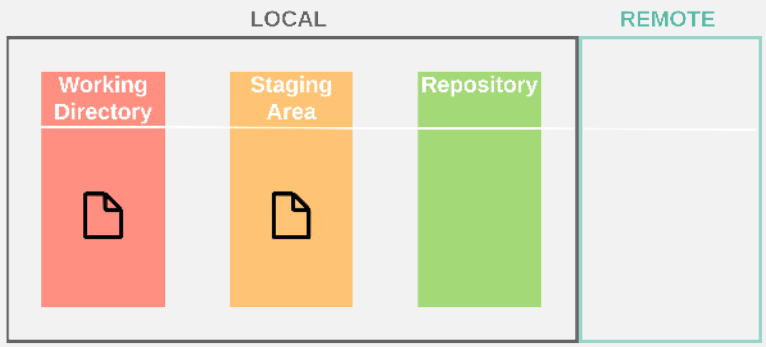
Bash

```
git add archivo
git status
```

Salida en pantalla

```
En la rama master

No hay commits todavía

Cambios a ser confirmados:
  (usa "git rm --cached <archivo>..." para sacar del área de stage)
      nuevos archivos: archivo
```

Figura 10.6: `git add`



10.2.4 `commit`

El comando `git commit` se utiliza en Git para confirmar los cambios realizados en el área de preparación, creando un nuevo commit en el historial del repositorio. La sintaxis básica es la siguiente:

```
git commit [-a] [-m mensaje]
```

La opción `-a` del comando `git commit` permite hacer una confirmación agregando automáticamente al área de preparación todos los archivos *con seguimiento*.

Si no incluyes la opción `-m`, Git abrirá un editor de texto para que escribas un mensaje más largo.

Después de realizar una confirmación, los cambios confirmados se guardan de manera permanente en el historial del repositorio.

! Observación

Es importante proporcionar mensajes claros y concisos para que otros desarrolladores (o tú mismo en el futuro) comprendan el propósito de la confirmación.
También deberías realizar confirmaciones frecuentes y atómicas para tener un historial de cambios claro y manejable.

Ejemplo 10.6. Finalmente veamos que sucede si confirmamos los cambios del área de preparación:

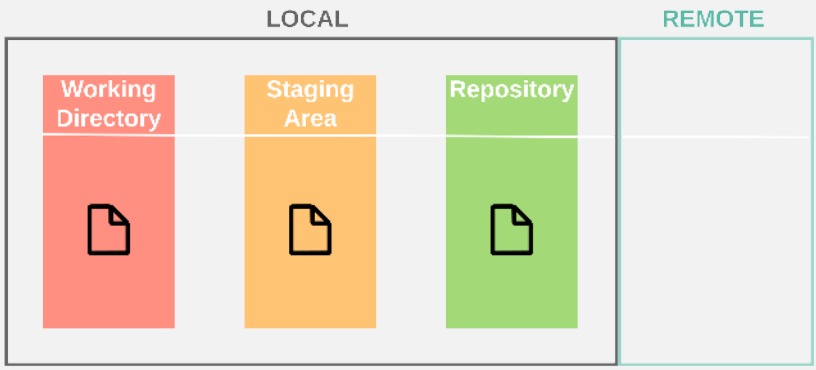
Bash

```
git commit -m "Primer commit."
git status
```

Salida en pantalla

```
[master (commit-raíz) 0287d2a] Primer commit.
1 file changed, 0 insertions(+), 0 deletions(-)
create mode 100644 archivo

En la rama master
nada para hacer commit, el árbol de trabajo está limpio
```

Figura 10.7: `git commit`



10.2.5 `log`

El comando `git log` se utiliza para visualizar el historial de confirmaciones de un repositorio. Proporciona información detallada sobre las confirmaciones realizados, como el autor, la fecha, el mensaje de la confirmación y el hash de la misma.

```
git log [opciones]
```

Las opciones mas importantes del comando son las siguientes:

`--oneline` Muestra cada confirmación en una sola línea, resumido con el hash y el mensaje del commit.

`--graph` Muestra una representación gráfica de las ramas y fusiones.

`-n` Limita la salida a un número específico de confirmaciones.

Ejemplo 10.7. Si observamos el registro de nuestro repositorio de prueba veremos lo siguiente:

Bash

```
git log
```

Salida en pantalla

```
commit c78ce51fb5e7a5f57b6dca800ce6a5455f8b1aae (HEAD ->  master)
Author: Damian Ariel <damarotte@gmail.com>
Date: Sat Jan 20 20:06:32 2024 -0300

    Primer commit.
```

En la salida podemos observar una suma de verificación en color amarillo. Este, será un identificador único que podemos utilizar para hacer referencia a dicha confirmación.

! Observación

En la salida del comando también puede observarse que esta es la última confirmación de la rama *master* (en verde) y que actualmente estamos trabajando en dicha rama (en celeste).

Ejemplo 10.8. También podemos ver la misma información en forma reducida:

Bash

```
git log --oneline
```

Salida en pantalla

```
c78ce51 (HEAD ->  master) Primer commit.
```

10.2.6 `restore`

El comando `git restore` se utiliza para restaurar archivos en el directorio de trabajo a un estado específico. También puede revertir cambios realizados en el área de preparación y deshacer modificaciones no deseadas. Aquí está la sintaxis básica del comando:

```
git restore [opciones] archivo
```

Las opciones mas comunes son las siguientes:

`-S` Restaura los archivos en el área de preparación al estado de la última confirmación.

`-s` Permite especificar la fuente desde la que se restauraran los archivos.

Si se utiliza sin opciones, se restauraran archivos en el directorio de trabajo desde el área de preparación.

! Observación

`git restore` no afecta el historial de confirmaciones, solo revierte los cambios en el área de preparación o el directorio de trabajo.

Ejemplo 10.9. Veamos ahora la salida de `git status` si borramos el archivo del directorio de trabajo:

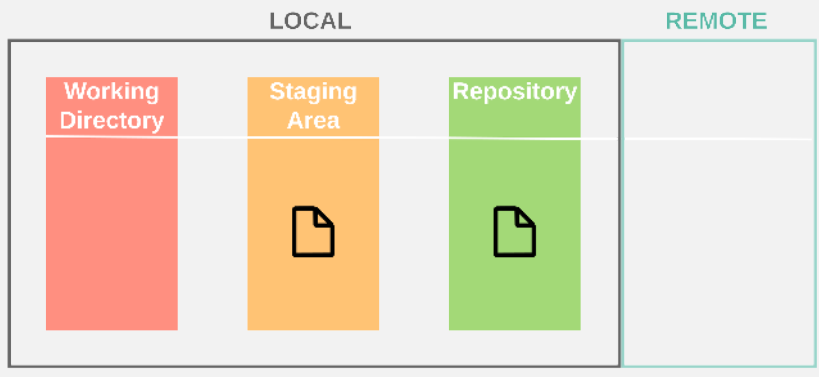
Bash

```
rm archivo
git status
```

Salida en pantalla

```
En la rama master
Cambios no rastreados para el commit:
  (usa "git add/rm <archivo>..." para actualizar a lo que se le va a
hacer commit)
  (usa "git restore <archivo>..." para descartar los cambios en el
directorio de trabajo)
        borrados: archivo

sin cambios agregados al commit (usa "git add" y/o "git commit -a")
```

Figura 10.8: `rm`



Ejemplo 10.10. Si recuperamos el archivo borrado a partir de la versión confirmada en el repositorio observaremos lo siguiente:

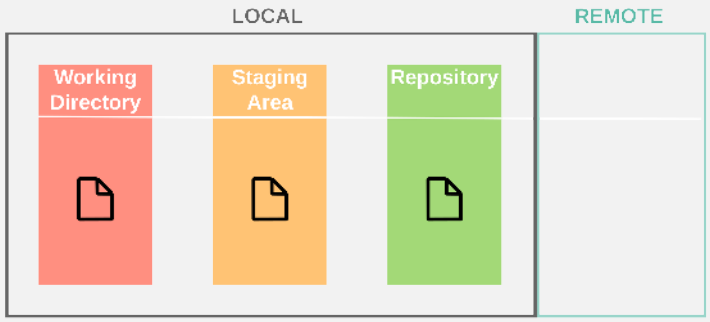
Bash

```
git restore archivo
git status
```

Salida en pantalla

```
En la rama master
nada para hacer commit, el árbol de trabajo está limpio
```

Figura 10.9: `git restore`

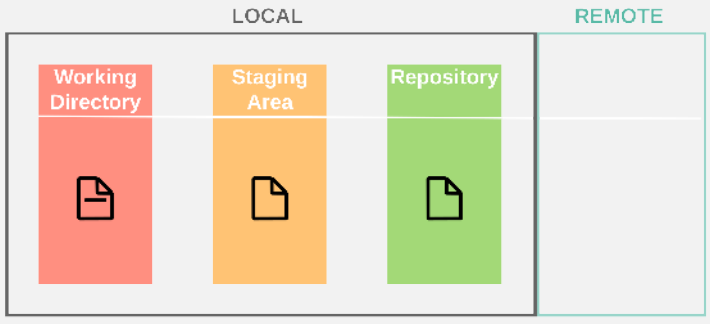


Ejemplo 10.11. Agreguemos contenido al área de preparación:

Bash

```
echo contenido > archivo
```

Figura 10.10: Agregando contenido al directorio de trabajo



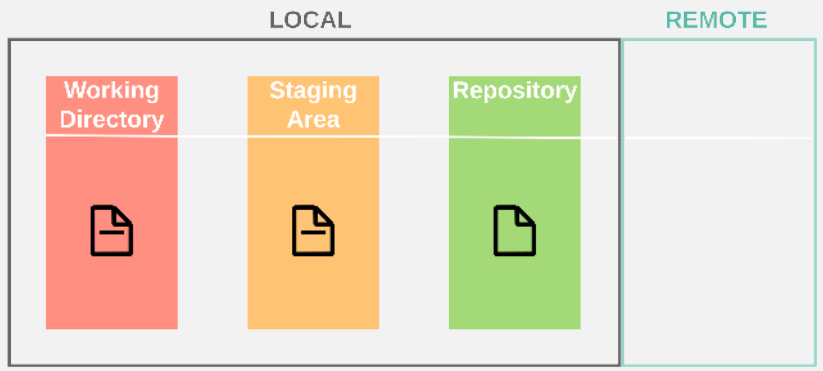
Bash

```
git add archivo
git status
```

Salida en pantalla

```
En la rama master
Cambios a ser confirmados:
  (usa "git restore --staged <archivo>..." para sacar del área de stage)
        modificados: archivo
```

Figura 10.11: `git add`



Ejercicio 10.12. Veamos ahora que sucede si restauramos el archivo en el área de preparación, a la versión de la última confirmación:

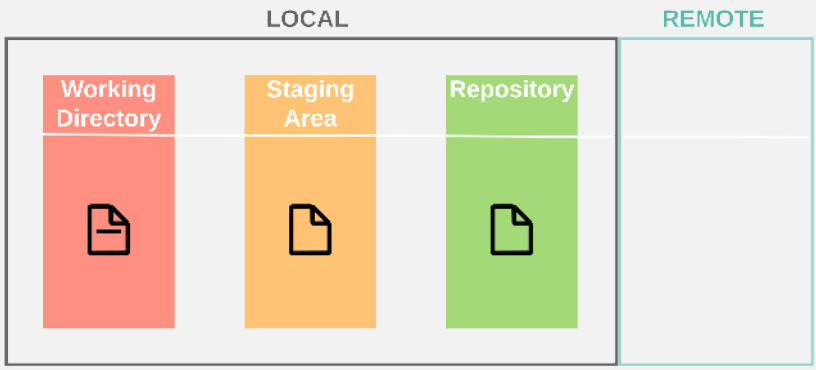
Bash

```
git restore --staged archivo
git status
```

Salida en pantalla

```
En la rama master
Cambios no rastreados para el commit:
  (usa "git add <archivo>..." para actualizar lo que será confirmado)
  (usa "git restore <archivo>..." para descartar los cambios en el
directorio de trabajo)
        modificados: archivo

sin cambios agregados al commit (usa "git add" y/o "git commit -a")
```

Figura 10.12: `git restore --staged`



10.2.7 `diff`

El comando `git diff` en Git se utiliza para mostrar las diferencias entre diferentes estados del proyecto. Principalmente, compara el contenido entre el directorio de trabajo y el área de preparación, o entre dos confirmaciones, ramas o cualquier otro punto en el historial de cambios.

Su sintaxis es la siguiente:

```
git diff [--staged | commit1 commit2] [archivo]
```

- Si usamos el comando sin opciones, muestra las diferencias entre los archivos que has modificado en el directorio de trabajo y los que ya has añadido al área de preparación.
- Si agregamos la opción `--staged`, muestra las diferencias entre los archivos en el área de preparación y la última confirmación realizada.
- También podemos comparar el contenido entre dos confirmaciones especificando sus respectivas sumas de verificación.

Ejemplo 10.13. Observemos las diferencias entre el directorio de trabajo y el área de preparación, luego de agregar contenido a un archivo:

Bash

```
git diff
```

Salida en pantalla

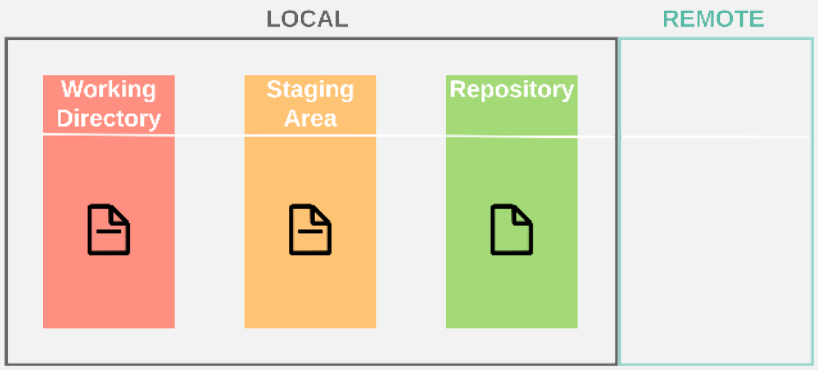
```
diff --git a/archivo b/archivo
index e69de29..764df4e 100644
--- a/archivo
+++ b/archivo
@@ -0,0 +1 @@
+contenido
```

Ejemplo 10.14. Agreguemos la nueva versión al área de preparación:

Bash

```
git add archivo
```

Figura 10.13: `git add`



Ahora modifiquemos el contenido de nuestro archivo. La salida ahora sera algo similar a esto:

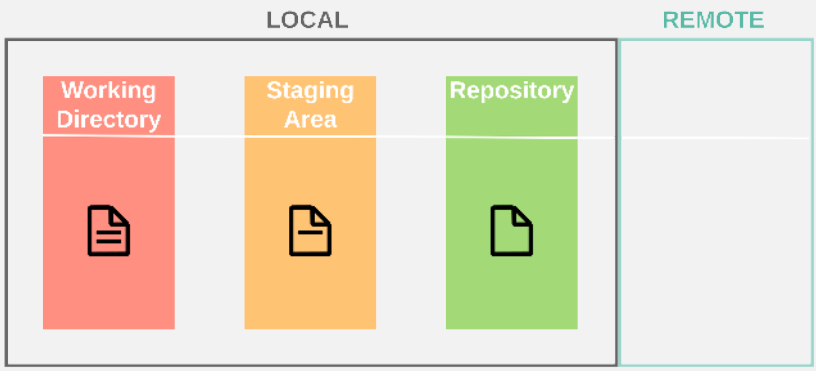
Bash

```
echo mas contenido >> archivo && git diff
```

Salida en pantalla

```
diff --git a/archivo b/archivo
index 764df4e..8c064c0 100644
--- a/archivo
+++ b/archivo
@@ -1 +1,2 @@
contenido
+mas contenido
```

Figura 10.14: Agregando mas contenido al archivo



Ejemplo 10.15. Si comparamos el área de preparación con la última confirmación veremos algo como esto:

Bash

```
git diff --staged
```

Salida en pantalla

```
diff --git a/archivo b/archivo
index e69de29..764df4e 100644
--- a/archivo
+++ b/archivo
@@ -0,0 +1 @@
+contenido
```

10.2.8 ~~reset~~

10.2.9 Ejercicios

Ejercicio 10.16. Investigue para que sirve la opción «`-a`» del comando «`git add`».

Ejercicio 10.17. Investigue para que sirve el comando «`git revert`».

Ejercicio 10.18. Investigue para que sirve el comando «`git blame`».

Ejercicio 10.19. Investigue para que sirve el comando «`git show`».

10.3 Trabajando con ramas

10.3.1 Definición

Una rama es una línea de desarrollo independiente que permite trabajar en nuevas características, correcciones de errores o experimentos sin afectar el estado principal del proyecto. Cada rama en un repositorio Git representa una línea de tiempo separada de confirmaciones que evoluciona de forma paralela a otras ramas.

! Observación

Internamente una rama es simplemente un nombre que le ponemos a una confirmación. Esta referencia se va actualizando con cada confirmación nueva que agregamos a la rama activa.

Git posee un puntero de referencia especial llamado «*HEAD*» que representa la rama actual. Puedes pensar en «*HEAD*» como el «punto de vista actual» o el «punto de referencia» en el historial de tu proyecto.

10.3.2 `branch`

El comando `git branch` se utiliza para realizar operaciones relacionadas con ramas en tu repositorio. Puedes listar las ramas existentes y crear nuevas ramas.

Su sintaxis es la siguiente:

```
git branch [-a | nombre]
```

- Si se utiliza sin ninguna opción, se listaran todas las ramas locales.
- También puede agregarse la opción «`-a`» para ver las ramas remotas.
- Para crear una rama, basta con pasarle como argumento su nombre.

! Observación

Debemos notar que cuando creamos una nueva rama el puntero de referencia no se mueve automáticamente, es decir, seguimos posicionados en la rama original.

Ejemplo 10.20. Para ver las ramas locales usamos el comando sin opciones:

Bash

```
git branch
```

Salida en pantalla

```
* master
```

Ejemplo 10.21. Creamos una «nueva» rama:

Bash

```
git branch nueva
git branch
```

Salida en pantalla

```
* master
  nueva
```

10.3.3 `switch`

El comando `git switch` se utiliza para cambiar entre ramas existentes. Simplemente basta escribir el comando seguido del nombre de la rama a la cual se pretende cambiar.

Observación

A partir de Git 2.23, se recomienda usar `git switch` en lugar de `git checkout` para cambiar entre ramas, ya que es más fácil de entender y menos propenso a errores.

Ejemplo 10.22. Veamos que sucede si cambiamos a la nueva rama:

Bash

```
git switch nueva
git log
```

Salida en pantalla

```
M archivo
Cambiado a rama 'nueva'

commit 980bc28d5e306ae1fad33b9731126b4964813fc4 (HEAD ->  nueva, master)
Author: Damian Ariel Marotte <damarotte@gmail.com>
Date: Mon Jan 22 18:37:09 2024 -0300
    Primer commit.
```

Si observamos con atención, comprenderemos que por el momento ambas ramas son idénticas pero que la rama activa es la «nueva» rama.

Ejemplo 10.23. Confirmemos ahora los cambios que previamente preparamos y observemos el estado del repositorio:

Bash

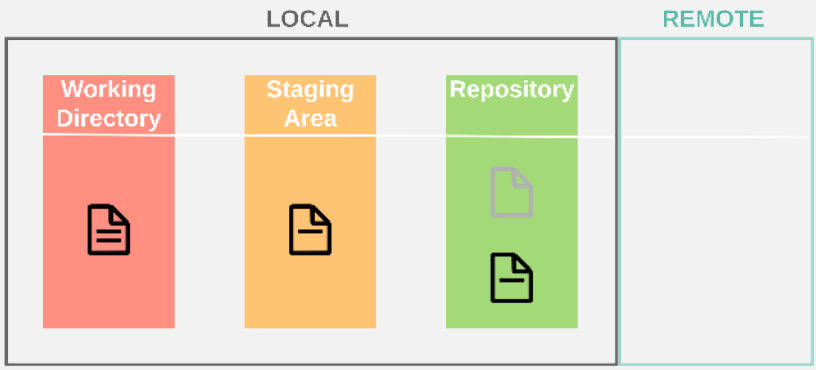
```
git commit -m "Segundo commit."
git log
cat archivo
```

Salida en pantalla

```
[nueva 22df02d] Segundo commit.
1 file changed, 1 insertion(+)

commit 22df02db5a9558bd7748c5d403cdf0ac836d8897 (HEAD ->  nueva)
Author: Damian Ariel Marotte <damarotte@gmail.com>
Date: Mon Jan 22 19:01:52 2024 -0300
    Segundo commit.

commit 980bc28d5e306ae1fad33b9731126b4964813fc4 (master)
Author: Damian Ariel Marotte <damarotte@gmail.com>
Date: Mon Jan 22 18:37:09 2024 -0300
    Primer commit.

contenido
mas contenido
```

Nótese que el contenido del archivo difiere de la confirmación, pues en el momento en el que se agrego al área de preparación no incluía «mas contenido».

Figura 10.15: `git commit`



Ejemplo 10.24. Observemos que sucede si volvemos a la rama original:

Bash

```
git switch master
```

Salida en pantalla

```
error: Los cambios locales de los siguientes archivos serán sobrescritos
por checkout:
        archivo
Por favor realiza un commit con los cambios o un stash antes de cambiar
de rama.
Abortando
```

Git sabe que nuestro archivo en el directorio de trabajo tiene información que no está en la otra rama, y que el cambio de rama haría que esa información se pierda. Agreguemos esos cambios a una confirmación y volvamos a intentarlo:

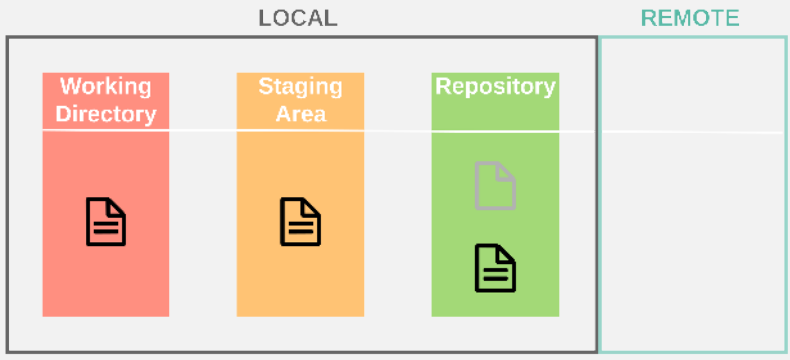
Bash

```
git commit -a -m "Tercer commit."
```

Salida en pantalla

```
[nueva bab8a34] Tercer commit.
1 file changed, 1 insertion(+)
```

Figura 10.16: `git commit`



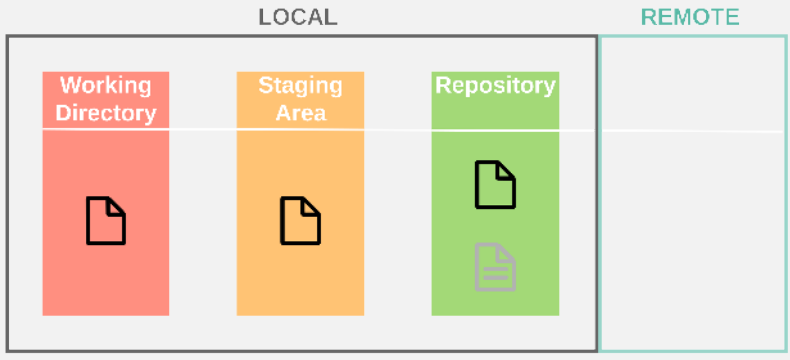
Bash

```
git switch master
```

Salida en pantalla

```
Cambiado a rama 'master'
```

Figura 10.17: `git switch`



! Observación

Vale la pena notar que si observamos el contenido de nuestro archivo, lo veremos vacío y no con los cambios confirmados en la nueva rama.

10.3.4 `merge`

El comando `git merge` en Git se utiliza para combinar los cambios de otra rama en la rama actual. Su sintaxis es simplemente:

```
git merge rama
```

Cuando escribimos `git merge rama` se incorporan en la rama actual, las confirmaciones de la rama indicada en el comando.

! Observación

En el proceso de fusión, Git intenta combinar automáticamente los cambios. Sin embargo, pueden surgir conflictos que requerirán intervención manual. En esos casos, Git marca los archivos conflictivos y te pedirá que resuelvas los conflictos antes de hacer la fusión.

Ejemplo 10.25. Incorporemos en nuestra rama actual, el contenido de nuestro archivo de la rama «nueva»:

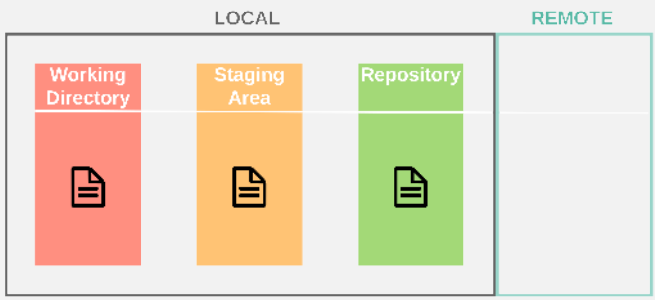
Bash

```
git merge nueva
```

Salida en pantalla

```
Actualizando 0287d2a..bab8a34
Fast-forward
 archivo | 2 ++
 1 file changed, 2 insertions(+)
```

Figura 10.18: `git merge`



Bash

```
git log
```

Salida en pantalla

```
commit bab8a343850d20cb4ca72c1264acab03935f9d96 (HEAD ->  master, nueva)
Author: Damian Ariel Marotte <damarotte@gmail.com>
Date: Tue Jan 23 06:57:16 2024 -0300

    Tercer commit.

commit f4e744110d15d2879a7b7532fd806aa6038c9970
Author: Damian Ariel Marotte <damarotte@gmail.com>
Date: Tue Jan 23 06:48:31 2024 -0300

    Segundo commit.

commit 0287d2a84ab9a12e02a5ac052cc5b797525b2de3
Author: Damian Ariel Marotte <damarotte@gmail.com>
Date: Tue Jan 23 05:50:05 2024 -0300

    Primer commit.
```

10.3.5 ~~*rebase*~~

10.3.6 ~~*Resolución de conflictos*~~

10.3.7 ~~*Flujos de trabajos*~~

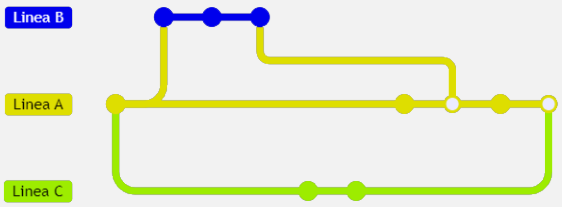
10.3.8 *Ejercicios*

Ejercicio 10.26. Investigue a que hace referencia el puntero «HEAD~n».

Ejercicio 10.27. ✔ Haga un repositorio con el siguiente historial:

Los ejercicios marcados con ✔ son ejercicios recomendados.

Figura 10.19: Lineas de subterraneo



10.4 Repositorios remotos

GitHub, GitLab y Bitbucket son plataformas de desarrollo colaborativo que utilizan Git como su sistema de control de versiones subyacente. Ofrecen servicios para la gestión del código fuente y el desarrollo de software como la Integración Continua (CI), Despliegue Continuo (CD), interfaces web que facilitan la colaboración, revisión de código, seguimiento de problemas y otras tareas relacionadas con el desarrollo de software.

10.4.1 `remote`

El comando `git remote` en Git se utiliza para gestionar conexiones remotas con repositorios remotos. Un repositorio remoto es una versión de tu proyecto que se almacena en otro lugar, ya sea en otro directorio en tu máquina local o en un servidor remoto.

Aquí está la sintaxis general del comando `git remote`:

```
git remote [-v | add nombre ruta | remove nombre]
```

- La opción `-v` es comúnmente utilizada para obtener una vista detallada de los repositorios remotos configurados.
- El comando `git remote add` agrega un repositorio remoto, mientras que `git remote remove` sirve para quitarlos.

Ejemplo 10.28. Creemos un repositorio vacío en Gitlab y configuremos dicho repositorio como fuente remota de nuestro repositorio local:

Bash

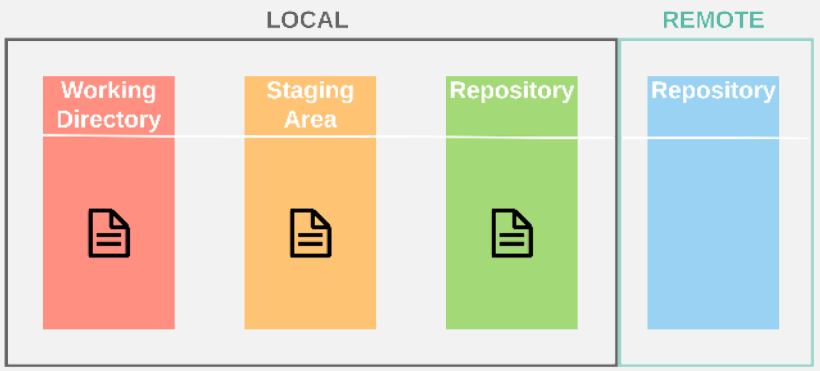
```
git remote add origin https://gitlab.com/damianarielm/borrar
git remote -v
git branch -a
```

Salida en pantalla

```
origin https://gitlab.com/damianarielm/borrar (fetch)
origin https://gitlab.com/damianarielm/borrar (push)

* master
  nueva
```

Por el momento, nuestra versión remota del repositorio permanece completamente vacía.

Figura 10.20: Repositorio remoto en gitlab



10.4.2 `push`

El comando `git push` se utiliza para enviar las confirmaciones locales de una rama a un repositorio remoto. Esta acción actualiza el estado de la rama en el repositorio remoto con los cambios locales realizados.

Aquí está la sintaxis general del comando:

```
git push [-u remoto rama]
```

- La opción `-u` configura la rama remota como la rama de seguimiento de la rama local. Esto significa que en futuros `git push` o `git pull`, Git sabrá a qué rama remota hacer referencia.
- Si se utiliza sin opciones, se enviaran los cambios a la rama remota previamente configurada.

Ejemplo 10.29. Ahora que sabemos como hacerlo, enviemos nuestra versión del repositorio al remoto que configuramos anteriormente:

Bash

```
git push -u origin master
git status
```

Salida en pantalla

```
advertencia: redirigiendo a https://gitlab.com/damianarielm/borrar.git/
Enumerando objetos: 9, listo.
Contando objetos: 100% (9/9), listo.
Compresión delta usando hasta 2 hilos
Comprimiendo objetos: 100% (3/3), listo.
Escribiendo objetos: 100% (9/9), 668 bytes | 668.00 KiB/s, listo.
Total 9 (delta 0), reusados 0 (delta 0), pack-reusados 0
To https://gitlab.com/damianarielm/borrar
 * [new branch] master ->  master
rama 'master' configurada para rastrear 'origin/master'.

En la rama master
Tu rama está actualizada con 'origin/master'.
nada para hacer commit, el árbol de trabajo está limpio
```

Bash

```
git branch -a
git log
```

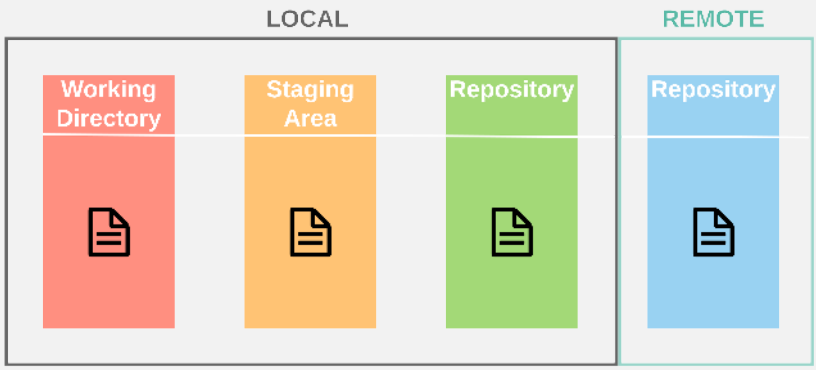
Salida en pantalla

```
* master
 nueva
 remotes/origin/master
```

Salida en pantalla

```
commit bab8a343850d20cb4ca72c1264acab03935f9d96 (HEAD ->  master,
origin/master, nueva)
Author: Damian Ariel Marotte <damarotte@gmail.com>
Date: Tue Jan 23 06:57:16 2024 -0300

    Tercer commit.

commit f4e744110d15d2879a7b7532fd806aa6038c9970
Author: Damian Ariel Marotte <damarotte@gmail.com>
Date: Tue Jan 23 06:48:31 2024 -0300

    Segundo commit.

commit 0287d2a84ab9a12e02a5ac052cc5b797525b2de3
Author: Damian Ariel Marotte <damarotte@gmail.com>
Date: Tue Jan 23 05:50:05 2024 -0300

    Primer commit.
```

Figura 10.21: `git push`



10.4.3 `fetch`

El comando `git fetch` en Git se utiliza para recuperar los cambios y referencias de un repositorio remoto en tu repositorio local.

Aquí está la sintaxis general del comando:

```
git fetch [remoto] [rama]
```

- El primer argumento especifica el nombre del repositorio remoto del que deseas recuperar los cambios.
- El segundo argumento especifica la rama remota de la cual deseas recuperar los cambios. Si no se especifica una rama remota, Git recuperará todas las ramas remotas.

! Observación

`git fetch` solo descarga los cambios del repositorio remoto, sin fusionarlos automáticamente con tu trabajo actual.

Ejemplo 10.30. Creemos una última confirmación desde Gitlab y observemos el estado del repositorio:

Bash

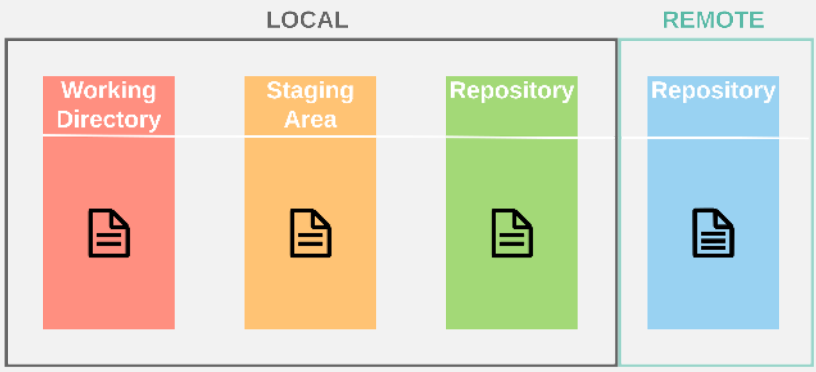
```
git status
```

Salida en pantalla

```
En la rama master
Tu rama está actualizada con 'origin/master'.
nada para hacer commit, el árbol de trabajo está limpio
```

Git nos indica que nuestra rama local está sincronizada con la última versión conocida del repositorio remoto.

Figura 10.22: Creando una confirmación desde Gitlab



Para actualizar esta información debemos traer la nueva versión desde Gitlab:

Bash

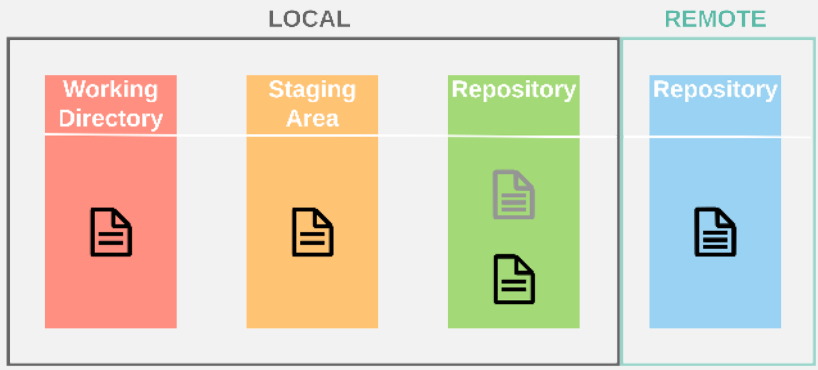
```
git fetch
git status
```

Salida en pantalla

```
En la rama master
Tu rama está detrás de 'origin/master' por 1 commit, y puede ser
avanzada rápido.
  (usa "git pull" para actualizar tu rama local)
nada para hacer commit, el árbol de trabajo está limpio
```

Ahora si podemos observar que nuestra rama local carece de la confirmación que tiene nuestra versión remota de la rama.

Figura 10.23: `git fetch`



Si lo deseamos, podemos integrar estos cambios mediante una fusión:

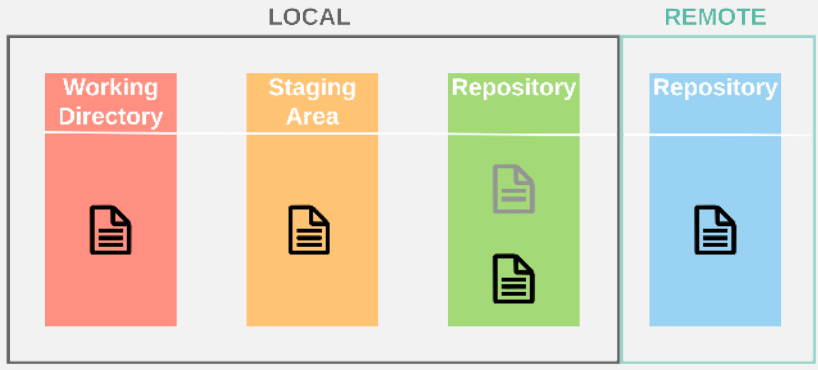
Bash

```
git merge origin/master
```

Salida en pantalla

```
Actualizando bab8a34..01090d3
Fast-forward
archivo | 1 +
1 file changed, 1 insertion(+)
```

Figura 10.24: `git merge`



Bash

```
git status
```

Salida en pantalla

```
En la rama master
Tu rama está actualizada con 'origin/master'.
nada para hacer commit, el árbol de trabajo está limpio
```

10.4.4 `pull`

El comando `git pull` se utiliza para recuperar los cambios de un repositorio remoto y fusionarlos automáticamente con la rama local actual. En esencia, `git pull` combina dos operaciones: `git fetch`, que descarga los cambios del repositorio remoto a tu repositorio local, y `git merge`, que fusiona esos cambios en tu rama local actual.

Aquí está la sintaxis general del comando:

```
git pull [remoto] [rama]
```

- El primer argumento especifica el nombre del repositorio remoto del que deseas recuperar los cambios.
- El segundo argumento especifica la rama remota de la cual deseas recuperar los cambios. Si no se especifica una rama remota, Git recuperará todas las ramas remotas.

10.4.5 `clone`

El comando `git clone` en Git se utiliza para crear una copia local de un repositorio remoto. Esta copia local incluye todos los archivos, ramas, historial de confirmaciones y configuraciones del repositorio remoto. Es una forma común de comenzar a trabajar en un proyecto nuevo o colaborar en un proyecto existente.

Aquí está la sintaxis general del comando:

```
git clone ruta [directorio] [--depth numero]
```

Si no especificas un nombre de directorio local, Git utilizará el nombre del repositorio remoto como nombre del directorio.

La opción `--depth` se utiliza para clonar con una profundidad limitada de historial de confirmaciones. Específicamente, te permite especificar cuántas confirmaciones de historial de la rama principal deseas clonar.

! Observación

En esencia, `git clone` es la combinación de cuatro comandos: `git init`, `git remote add`, `git pull` y `git switch`.

10.4.6 ~~Fork~~

10.4.7 ~~Pull request~~

10.4.8 ~~LFS~~

11

~~Contenedores~~

11.1 ~~Simulación~~

11.2 ~~Emulación~~

11.3 ~~Virtualización~~

11.4 ~~Contenedores~~

12

Otros conceptos

12.1 venv

12.2 chroot

12.3 Colab

Apéndice

13

Instalación de linux

13.1 Máquina virtual

13.1.1 VirtualBox

«VirtualBox» es un software de virtualización de código abierto desarrollado por Oracle. Permite instalar y ejecutar sistemas operativos diferentes, como versiones de Windows, Linux, macOS y otros, dentro de una ventana de software en tu sistema operativo principal.

Antes de instalar Linux en la máquina virtual debemos descargar una distribución desde internet. En la materia utilizaremos «Lubuntu».

Creación de una máquina virtual

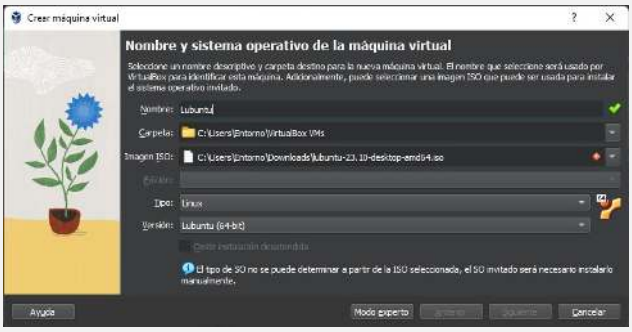
Para crear una nueva máquina virtual elegimos la opción «*Nueva*» en la pantalla principal.

Figura 13.1: Pantalla principal de VirtualBox

Configuración del SO

A continuación elegimos la ubicación dentro del sistema de archivos, así como también la distribución que vamos a instalar y la imagen de disco que descargamos.

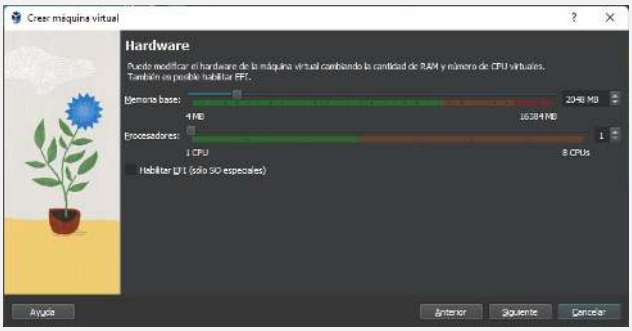
Figura 13.2: Pantalla principal de VirtualBox



Elección del hardware

En la siguiente ventana es posible elegir cuanta memoria y procesadores utilizará la máquina virtual. Se recomienda un mínimo de 2GB.

Figura 13.3: Configuración del hardware



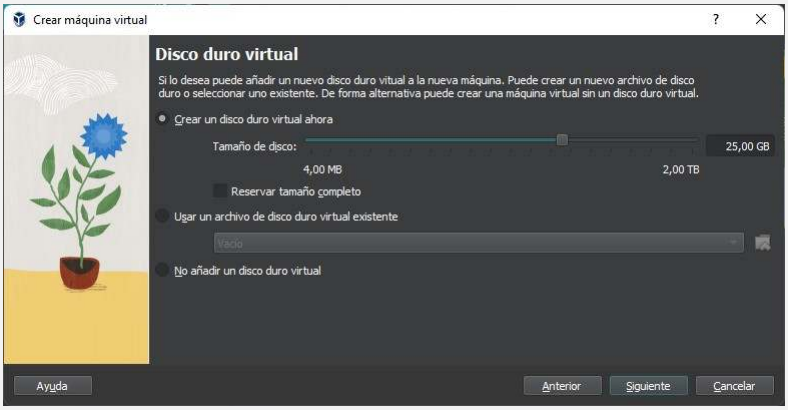
! Observación

Vale la pena notar que estos recursos no estarán disponibles para la máquina anfitriona.

Creación de un disco virtual

Ahora debemos elegir la capacidad del disco de la máquina virtual. Recomendamos 15GB como mínimo.

Figura 13.4: Creación de un disco virtual



Último paso

Finalmente podemos hacer click en «*Terminar*» y ya estaremos listos para usar nuestra máquina virtual.

Figura 13.5: Resumen

13.1.2 *Lubuntu*

Inicio del LiveCD

Una vez que tengamos configurada nuestra máquina virtual con el correspondiente CD de «Lubuntu» insertado, podemos ejecutar la misma. Luego de algunos minutos veremos una pantalla como esta:

Figura 13.6: Inicio del LiveCD

Llegados a este punto, elegimos el idioma y ya podemos dar comienzo a la instalación haciendo click en «*Install Lubuntu*».

Comienzo de la instalación

A continuación elegimos el idioma de preferencia y presionamos el botón «*Siguiente*».

Figura 13.7: Comienzo de la instalación

Configuración regional

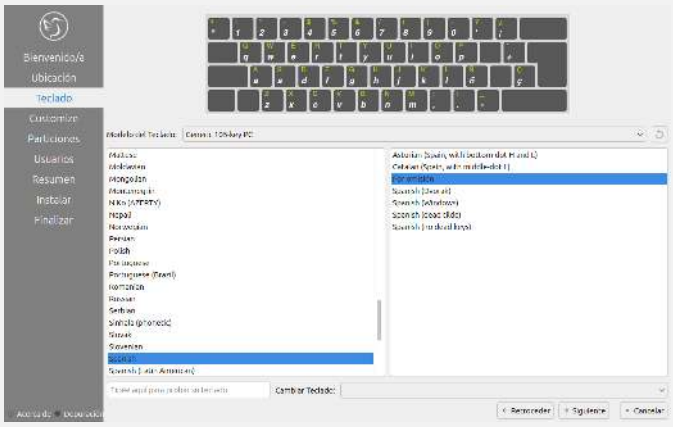
Ha llegado la hora de configurar la región en la que opera el dispositivo. El sistema operativo mostrará fechas, horarios y unidades de acuerdo a lo seleccionado en esta sección.

Figura 13.8: Configuración regional

Distribución del teclado

En esta sección podemos elegir la distribución del teclado. En el cuadro de texto de la zona inferior, podemos probar que las teclas se muestren bien.

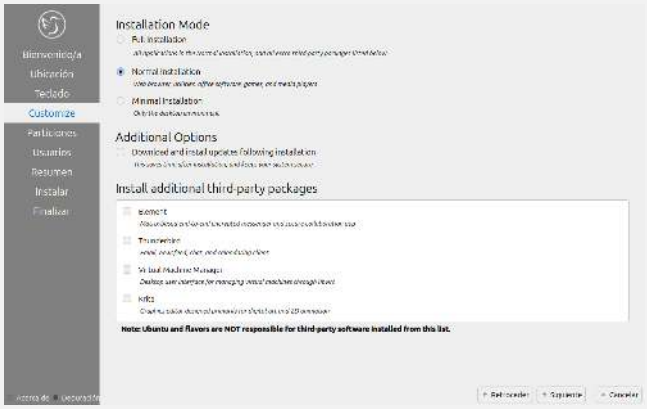
Figura 13.9: Distribución del teclado



Personalización de la instalación

Ahora podemos elegir si queremos una instalación mínima o una mas completa. Sugerimos elegir la opción que incluye el navegador, aunque también puede instalarse posteriormente.

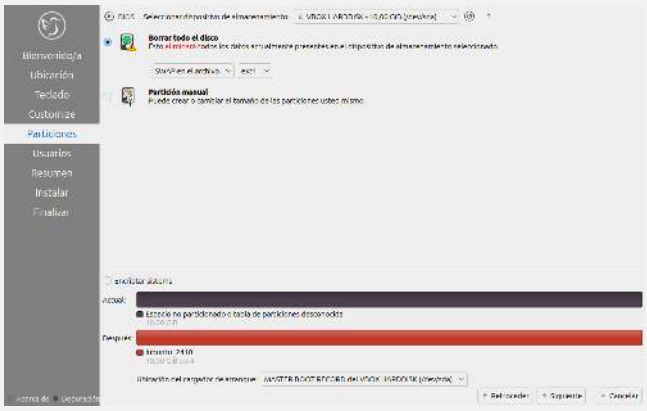
Figura 13.10: Personalización



Particionado del disco

Para nuestra primera instalación, elegiremos la opción de «*borrar el disco*» como esquema de particionado.

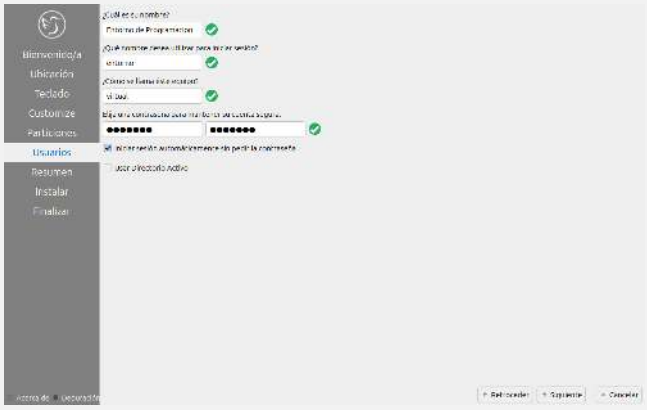
Figura 13.11: Particionado del disco



Creación del usuario principal

Ahora debemos elegir nuestro nombre, nombre de usuario, nombre del equipo y contraseña. También podemos marcar la casilla de verificación si deseamos iniciar sesión en la máquina virtual sin introducir la contraseña.

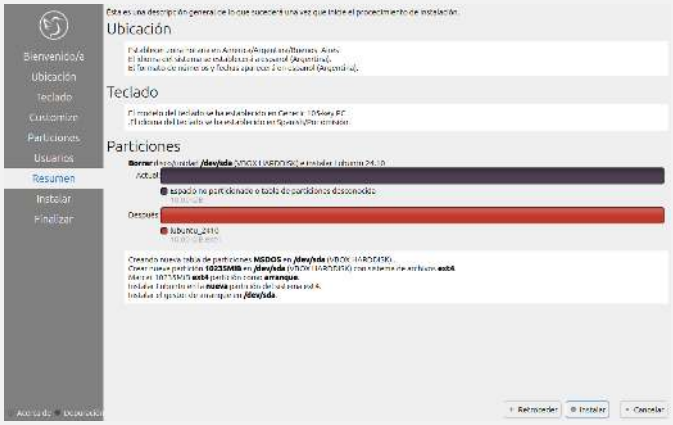
Figura 13.12: Creación del usuario principal



Resumen de los datos

Reunida toda la información de los pasos anteriores, podemos verificarla una vez mas antes de hacer click en «*Instalar*».

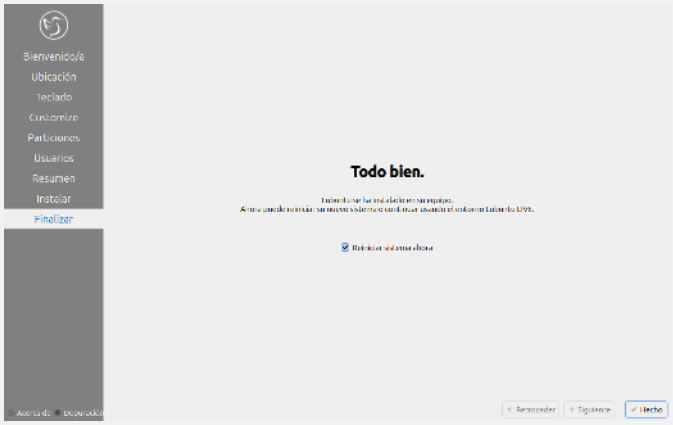
Figura 13.13: Resumen de los datos



Final de la instalación

Despues de varios minutos la instalación habrá finalizado, momento en el cual presionaremos el botón «*Hecho*».

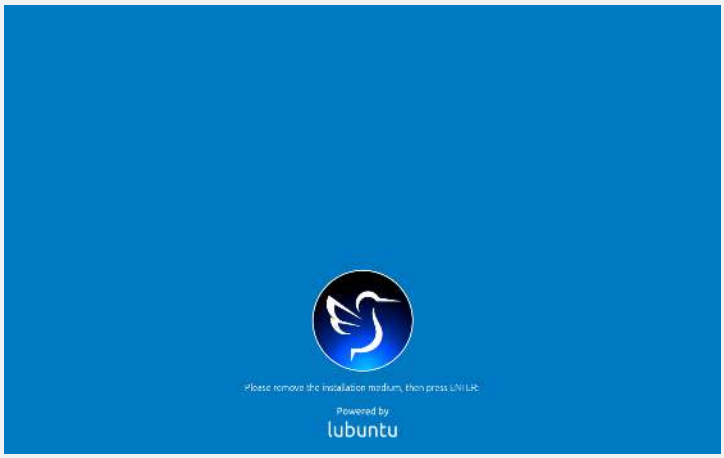
Figura 13.14: Final de la instalación



Reinicio de la máquina virtual

Finalmente, presionamos la tecla ⌨ Enter y luego reiniciamos la *máquina virtual*.

Figura 13.15: Reinicio de la máquina virtual



13.1.3 Guest Additions

Las «*Guest Additions*» son un conjunto de controladores y utilidades adicionales que se instalan dentro de una máquina virtual (VM) que se ejecuta en VirtualBox. Están diseñadas para mejorar la experiencia del usuario y la funcionalidad de la VM en general.

«Guest Addittions» *significa en ingles «Agregados del huesped».*

Algunas de las características y mejoras que proporcionan las Guest Additions incluyen:

Integración del ratón y el teclado Las Guest Additions permiten una integración más fluida del ratón y el teclado entre el sistema operativo anfitrión y el invitado.

Soporte de pantalla completa Permite habilitar el modo de pantalla completa para la VM para aprovechar al máximo el espacio de la pantalla y trabajar de manera más cómoda dentro de la máquina virtual.

integración del portapapeles Activa la funcionalidad del portapapeles entre el sistema operativo anfitrión y el invitado, lo que facilita el intercambio de texto y otros datos entre ambos sistemas.

Para instalarlas, basta con ejecutar los siguientes comandos:

Bash
```
sudo apt update
sudo apt install virtualbox-guest-x11
```

13.2 Instalación en pendrive

Para poder instalar *Linux* en un pendrive, necesitaremos alguna herramienta auxiliar como puede ser «*Rufus*». Debemos también tener disponible un pendrive de al menos 16GB, preferentemente USB 3.0.

Precacución

Todos los datos que se encuentren en el pendrive, se perderan tras la instalación.

Configuración

Una vez ejecutado *Rufus*, debemos configurar la instalación. En el primer cuadro desplegable, seleccionamos el pendrive donde vamos a instalarlo y en el segundo, la imagen de disco de la distribución que vayamos a instalar.

Figura 13.16: Configuración de Rufus

Si deseamos que nuestro sistema sea persistente, es decir, que conserve los cambios tras un reinicio, debemos mover el deslizador correspondiente hacia la derecha.

! Observación

Si decidimos usar persistencia el sistema sera considerablemente mas lento. Recomendamos una instalación sin persistencia.

Confirmación

Una vez que estemos listos, presionamos «*EMPEZAR*» y luego «*Ok*».

⚠ Precaución

Todos los datos que se encuentren en el pendrive, se perderan tras la instalación.

Figura 13.17: Confirmación

Arranque

Finalmente, debemos reiniciar y elegir el menú de booteo. Esto se logra presionando una tecla específica durante la etapa POST del arranque de la PC. Dependiendo del modelo esta tecla puede ser: ⌨ Esc, ⌨ F9, ⌨ F11 o ⌨ F12 (entre otras).

13.3 Otras alternativas

13.3.1 Subsistema de Windows para Linux

«W Windows Subsystem for Linux» (WSL) es una característica de Windows que permite ejecutar un entorno de Linux directamente en Windows sin necesidad de una máquina virtual o un arranque dual.

WSL utiliza una capa de compatibilidad para traducir las llamadas del sistema de Linux a llamadas del sistema de *Windows*, lo que permite que las aplicaciones de *Linux* se ejecuten de manera nativa en el sistema operativo *Windows*.

Para instalar WSL basta con correr el siguiente comando en PowerShell:

PowerShell

```
wsl --install -d ubuntu
```

Luego se debe ejecutar «`wsl`» para poder utilizarlo.

13.3.2 Colab

Es posible utilizar un entorno de «🔗 *Google Colab*» para utilizar Bash. Para esto debemos seguir estos tres pasos:

1. Instalar el módulo de Python `colab-xterm`:

   </> Código

   ```
   !pip install colab-xterm
   ```

2. Cargar la extensión en Colab:

   </> Código

   ```
   %load_ext colabxterm
   ```

3. Ejecutar `xterm`:

</> Código

```
%xterm
```

13.3.3 ~~Replit~~

13.3.4 ~~CoCalc~~

13.3.5 ~~Termux~~

14

Resolución de problemas

14.1 Activar *IVT / AMD-V*

IVT (Intel Virtualization Technology) y AMD-V (AMD Virtualization) son tecnologías de virtualización desarrolladas por *Intel* y AMD, respectivamente. Ambas tecnologías están diseñadas para mejorar el rendimiento y la eficiencia de la virtualización en sus respectivas arquitecturas de procesadores.

Para activar estas características es necesario ingresar a la configuración de la placa madre. Esto se logra presionando una tecla específica durante la etapa POST del arranque de la PC. Dependiendo del modelo esta tecla puede ser: ⌨ F2, ⌨ F10 o ⌨ DEL (entre otras).

Una vez allí simplemente es cuestión de navegar entre los menús hasta ubicar la opción adecuada y habilitarla. Finalmente se deben guardar los cambios y reiniciar el equipo.

Figura 14.1: Interfaz UEFI

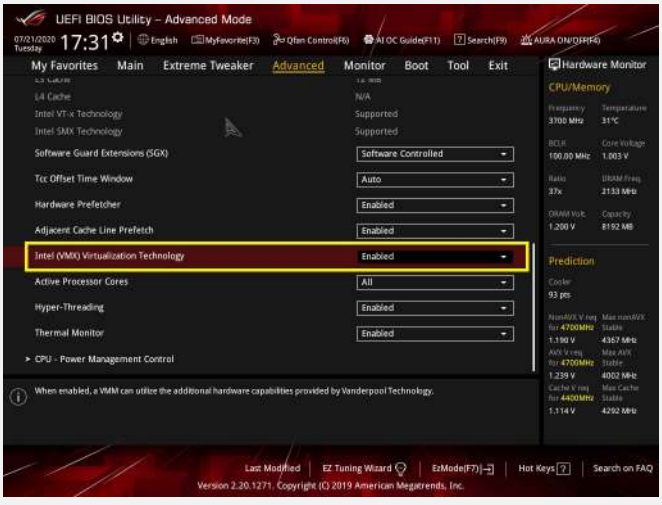
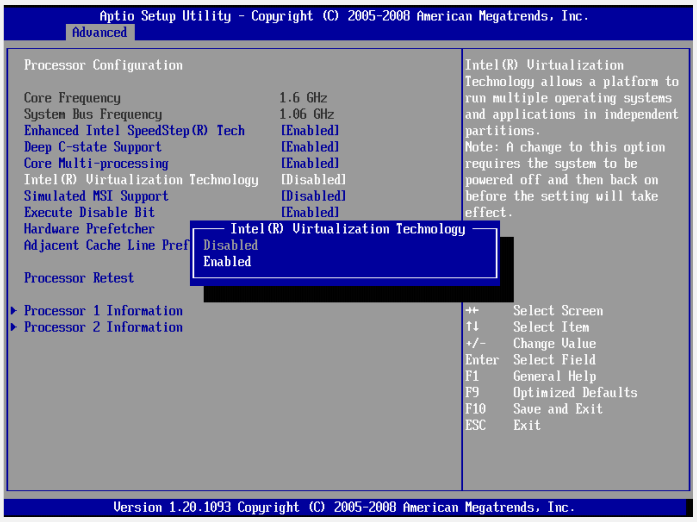


Figura 14.2: Interfaz BIOS



14.2 Librerías de C++

«*Visual C++ Redistributable Package*» es un conjunto de archivos y bibliotecas necesarios para que los programas desarrollados con Microsoft Visual C++ funcionen correctamente en una computadora. Cuando instalas un programa en *Windows* que fue creado con Visual C++, es posible que necesites instalar también estos «paquetes redistribuibles» si aún no están en tu sistema.

Puedes descargar la última versión de esta librería desde este enlace.

14.3 Poner man en español

Por defecto, la documentación de *Linux* se encuentra en idioma ingles. Para ponerla en idioma español debe utilizarse el siguiente comando:

Bash

```
sudo apt update
sudo apt install manpages-es
```

14.4 Actualizar Kernel para WSL

Si Windows no tiene actualizado el kernel de *Linux* para WSL, es probable que aparezca un error a la hora de instalar WSL:

Salida en pantalla

```
Installing, this may take a few minutes...
WslRegisterDistribution failed with error: 0x800701bc
Error: 0x800701bc WSL 2 requiere una actualizaci¾n de su
componente de kernel. Para obtener informaci¾n, visita
https://aka.ms/wsl2kernel
Press any key to continue...
```

Para solucionarlo, basta con actualizar el nucleo y volver a instalar:

PowerShell

```
wsl --update
wsl --install -d ubuntu
```

15

Programando un shell

15.1 Linea de comandos

Código

```python
import os

while True:
    comando = input()
    pid = os.fork()
    if pid == 0:
        os.execv(comando, [comando])
    else:
        os.waitid(os.P_PID, pid, os.WEXITED)
```

15.2 Variable PATH

</> Código

```python
import os

def buscar(comando):
    path = os.getenv("PATH")
    for directorio in path.split(":"):
        ruta = f"{directorio}/{comando}"
        if os.path.isfile(ruta):
            return ruta
    return comando

def ejecutar(programa):
    pid = os.fork()
    if pid == 0:
        os.execv(buscar(comando), [comando])
    else:
        os.waitid(os.P_PID, pid, os.WEXITED)

while True:
    comando = input()
    ejecutar(comando)
```

15.3 Manejo de argumentos

</> Código

```python
import os

def buscar(programa):
    path = os.getenv("PATH")
    for directorio in path.split(":"):
        ruta = f"{directorio}/{programa}"
        if os.path.isfile(ruta):
            return ruta
    return programa

def ejecutar(programa, argumentos):
    pid = os.fork()
    if pid == 0:
        os.execv(buscar(programa), argumentos)
    else:
        os.waitid(os.P_PID, pid, os.WEXITED)

def interpretar(comando):
    programa, *argumentos = comando.split()
    return programa, [programa] + argumentos

while True:
    comando = input()
    programa, argumentos = interpretar(comando)
    ejecutar(programa, argumentos)
```

15.4 Cambiar de directorio

</> Código

```python
import os

def buscar(programa):
    path = os.getenv("PATH")
    for directorio in path.split(":"):
        ruta = f"{directorio}/{programa}"
        if os.path.isfile(ruta):
            return ruta
    return programa

def ejecutar(programa, argumentos):
    pid = os.fork()
    if pid == 0:
        os.execv(buscar(programa), argumentos)
    else:
        os.waitid(os.P_PID, pid, os.WEXITED)

def interpretar(comando):
    comando = comando.replace("~", os.getenv("HOME"))
    programa, *argumentos = comando.split()
    return programa, [programa] + argumentos

def comando_cd(argumentos):
    if not argumentos[1:]:
        directorio = os.getenv("HOME")
    else:
        directorio = argumentos[1]
    os.chdir(directorio)

while True:
    comando = input()
    programa, argumentos = interpretar(comando)

    if programa == "cd":
        comando_cd(argumentos)
    else:
        ejecutar(programa, argumentos)
```

15.5 Prompt

</> Código

```python
import os

def prompt():
    cwd = os.getcwd()
    username = os.getlogin()
    hostname = os.uname().nodename
    print(f"{username}@{hostname}:{cwd}$", end =" ")
    return input()

def buscar(programa):
    path = os.getenv("PATH")
    for directorio in path.split(":"):
        ruta = f"{directorio}/{programa}"
        if os.path.isfile(ruta):
            return ruta
    return programa

def ejecutar(programa, argumentos):
    pid = os.fork()
    if pid == 0:
        os.execv(buscar(programa), argumentos)
    else:
        os.waitid(os.P_PID, pid, os.WEXITED)

def interpretar(comando):
    comando = comando.replace("~", os.getenv("HOME"))
    programa, *argumentos = comando.split()
    return programa, [programa] + argumentos

def comando_cd(argumentos):
    if not argumentos[1:]:
        directorio = os.getenv("HOME")
    else:
        directorio = argumentos[1]
    os.chdir(directorio)

while True:
    comando = prompt()
    programa, argumentos = interpretar(comando)

    if programa == "cd":
        comando_cd(argumentos)
    else:
        ejecutar(programa, argumentos)
```

15.6 Alias

</> Código

```
import os
def prompt():
    cwd = os.getcwd()
    username = os.getlogin()
    hostname = os.uname().nodename
    print(f"{username}@{hostname}:{cwd}$", end =" ")
    return input()
def buscar(programa):
    path = os.getenv("PATH")
    for directorio in path.split(":"):
        ruta = f"{directorio}/{programa}"
        if os.path.isfile(ruta):
            return ruta
    return programa
def ejecutar(programa, argumentos):
    pid = os.fork()
    if pid == 0:
        os.execv(buscar(programa), argumentos)
    else:
        os.waitid(os.P_PID, pid, os.WEXITED)
def interpretar(comando):
    comando = comando.replace("~", os.getenv("HOME"))
    programa, *argumentos = comando.split()
    if programa in alias:
        programa = alias[programa].split()[0]
        argumentos = alias[programa].split()[1:] + argumentos
    return programa, [programa] + argumentos
def comando_cd(argumentos):
    if not argumentos[1:]:
        directorio = os.getenv("HOME")
    else:
        directorio = argumentos[1]
    os.chdir(directorio)
def comando_alias(argumentos, alias):
    if not argumentos[1:]:
        print(alias)
    else:
        nombre, programa = argumentos[1].split("=")
        alias[nombre] = programa + " " + " ".join(argumentos[2:])
    return alias
alias = {}
while True:
    comando = prompt()
    programa, argumentos = interpretar(comando)
    if programa == "cd":
        comando_cd(argumentos)
    elif programa == "alias":
        comando_alias(argumentos, alias)
    else:
        ejecutar(programa, argumentos)
```

15.7 Segundo plano

</> Código

```python
import os

def prompt():
    cwd = os.getcwd()
    username = os.getlogin()
    hostname = os.uname().nodename
    print(f"{username}@{hostname}:{cwd}$", end =" ")
    return input()
def buscar(programa):
    path = os.getenv("PATH")
    for directorio in path.split(":"):
        ruta = f"{directorio}/{programa}"
        if os.path.isfile(ruta):
            return ruta
    return programa
def ejecutar(programa, argumentos, background):
    pid = os.fork()
    if pid == 0:
        os.execv(buscar(programa), argumentos)
    elif not background:
        os.waitid(os.P_PID, pid, os.WEXITED)
def interpretar(comando):
    background = comando[-1] == "&"
    if background: comando = comando[:-1]
    comando = comando.replace("~", os.getenv("HOME"))
    programa, *argumentos = comando.split()

    if programa in alias:
        programa = alias[programa].split()[0]
        argumentos = alias[programa].split()[1:] + argumentos

    return programa, [programa] + argumentos, background
def comando_cd(argumentos):
    if not argumentos[1:]:
        directorio = os.getenv("HOME")
    else:
        directorio = argumentos[1]
    os.chdir(directorio)
def comando_alias(argumentos, alias):
    if not argumentos[1:]:
        print(alias)
    else:
        nombre, programa = argumentos[1].split("=")
        alias[nombre] = programa + " " + " ".join(argumentos[2:])
    return alias

alias = {}
while True:
    comando = prompt()
    programa, argumentos, background = interpretar(comando)

    if programa == "cd":
        comando_cd(argumentos)
    elif programa == "alias":
        comando_alias(argumentos, alias)
    else:
        ejecutar(programa, argumentos, background)
```

15.8 Secuenciación

</> Código

```python
import os

def prompt():
    cwd = os.getcwd()
    username = os.getlogin()
    hostname = os.uname().nodename
    print(f"{username}@{hostname}:{cwd}$", end =" ")
    return input()
def buscar(programa):
    path = os.getenv("PATH")
    for directorio in path.split(":"):
        ruta = f"{directorio}/{programa}"
        if os.path.isfile(ruta):
            return ruta
    return programa
def ejecutar(programa, argumentos, background):
    pid = os.fork()
    if pid == 0:
        os.execv(buscar(programa), argumentos)
    elif not background:
        os.waitid(os.P_PID, pid, os.WEXITED)
def interpretar(comandos):
    for comando in comandos.split(";"):
        background = comando[-1] == "&"
        if background: comando = comando[:-1]
        comando = comando.replace("~", os.getenv("HOME"))
        programa, *argumentos = comando.split()

        if programa in alias:
            programa = alias[programa].split()[0]
            argumentos = alias[programa].split()[1:] + argumentos

        yield programa, [programa] + argumentos, background
def comando_cd(argumentos):
    if not argumentos[1:]:
        directorio = os.getenv("HOME")
    else:
        directorio = argumentos[1]
    os.chdir(directorio)
def comando_alias(argumentos, alias):
    if not argumentos[1:]:
        print(alias)
    else:
        nombre, programa = argumentos[1].split("=")
        alias[nombre] = programa + " " + " ".join(argumentos[2:])
    return alias

alias = {}
while True:
    comandos = prompt()

    for programa, argumentos, background in interpretar(comandos):
        if programa == "cd":
            comando_cd(argumentos)
        elif programa == "alias":
            comando_alias(argumentos, alias)
        else:
            ejecutar(programa, argumentos, background)
```

15.9 Exit

</> Código

```
import os

def prompt():
    cwd = os.getcwd()
    username = os.getlogin()
    hostname = os.uname().nodename
    print(f"{username}@{hostname}:{cwd}$", end =" ")
    return input()
def buscar(programa):
    path = os.getenv("PATH")
    for directorio in path.split(":"):
        ruta = f"{directorio}/{programa}"
        if os.path.isfile(ruta):
            return ruta
    return programa
def ejecutar(programa, argumentos, background):
    pid = os.fork()
    if pid == 0:
        os.execv(buscar(programa), argumentos)
    elif not background:
        os.waitid(os.P_PID, pid, os.WEXITED)
def interpretar(comandos):
    for comando in comandos.split(";"):
        background = comando[-1] == "&"
        if background: comando = comando[:-1]
        comando = comando.replace("~", os.getenv("HOME"))
        programa, *argumentos = comando.split()

        if programa in alias:
            programa = alias[programa].split()[0]
            argumentos = alias[programa].split()[1:] + argumentos

        yield programa, [programa] + argumentos, background
def comando_cd(argumentos):
    if not argumentos[1:]:
        directorio = os.getenv("HOME")
    else:
        directorio = argumentos[1]
    os.chdir(directorio)
def comando_alias(argumentos, alias):
    if not argumentos[1:]:
        print(alias)
    else:
        nombre, programa = argumentos[1].split("=")
        alias[nombre] = programa + " " + " ".join(argumentos[2:])
    return alias

alias = {}
comandos = ""
while comandos != "exit":
    comandos = prompt()

    for programa, argumentos, background in interpretar(comandos):
        if programa == "cd":
            comando_cd(argumentos)
        elif programa == "alias":
            comando_alias(argumentos, alias)
        elif programa != "exit":
            ejecutar(programa, argumentos, background)
```

15.10 Historial

</> Código

```python
import os
def prompt():
    cwd = os.getcwd()
    username = os.getlogin()
    hostname = os.uname().nodename
    print(f"{username}@{hostname}:{cwd}$", end =" ")
    return input()
def buscar(programa):
    path = os.getenv("PATH")
    for directorio in path.split(":"):
        ruta = f"{directorio}/{programa}"
        if os.path.isfile(ruta): return ruta
    return programa
def ejecutar(programa, argumentos, background):
    pid = os.fork()
    if pid == 0:
        os.execv(buscar(programa), argumentos)
    elif not background:
        os.waitid(os.P_PID, pid, os.WEXITED)
def interpretar(comandos, historial):
    for comando in comandos.split(";"):
        background = comando[-1] == "&"
        if background: comando = comando[:-1]
        comando = comando.replace("~", os.getenv("HOME"))
        comando = comando.replace("!!", historial[-1])
        if comando[0] == "!": comando = historial[int(comando[1:])]
        programa, *argumentos = comando.split()
        if programa in alias:
            programa = alias[programa].split()[0]
            argumentos = alias[programa].split()[1:] + argumentos
        yield programa, [programa] + argumentos, background
def comando_cd(argumentos):
    if not argumentos[1:]:
        directorio = os.getenv("HOME")
    else:
        directorio = argumentos[1]
    os.chdir(directorio)
def comando_alias(argumentos, alias):
    if not argumentos[1:]:
        print(alias)
    else:
        nombre, programa = argumentos[1].split("=")
        alias[nombre] = programa + " " + " ".join(argumentos[2:])
    return alias
alias = {}
comandos = ""
historial = []
while comandos != "exit":
    comandos = prompt()
    if comandos[0] != "!": historial.append(comandos)
    for programa, argumentos, background in interpretar(comandos, historial):
        if programa == "cd":
            comando_cd(argumentos)
        elif programa == "alias":
            comando_alias(argumentos, alias)
        elif programa == "history":
            for n, comando in enumerate(historial):
                print(f"{n} {comando}")
        elif programa != "exit":
            ejecutar(programa, argumentos, background)
```

Bibliografía

[1] Paul E. Ceruzzi - Computing: A Concise History.

[2] Brian Ward - How Linux Works.

[3] Daniel J. Barret - Efficient Linux at the Command Line.

[4] Anna Skoulikari - Learning Git.