Fundamentos de sistemas operativos. Entornos de trabajo

Book · July 2018

DOI: 10.16925/9789587601077

1 author:

Mateo Lezcano
Universidad Cooperativa de Colombia
41 PUBLICATIONS **34** CITATIONS

SEE PROFILE

FUNDAMENTOS DE SISTEMAS OPERATIVOS

ENTORNOS DE TRABAJO

MATEO G. LEZCANO BRITO

Fundamentos de sistemas operativos

Entornos de trabajo

Fundamentals of Operating Systems

Work Environments

RESUMEN

Este libro se dedica al estudio de los sistemas operativos desde la perspectiva de una carrera de perfil informático. El libro comienza con una introducción que ofrece una visión general de los sistemas operativos y luego se organiza en dos secciones. En la primera sección, el primer capítulo trata acerca de las formas de planificación del procesador central, y explica los diferentes tipos de planificadores y los algoritmos que se usan para distribuir el tiempo de procesamiento; también se analizan los problemas de sincronización entre procesos concurrentes. El segundo capítulo se basa en el concepto abstracto de archivo para explicar la forma en que se organiza la información sobre los soportes de almacenamiento externos; se discuten las distintas maneras de organizar el espacio ocupado por los archivos y el control sobre los espacios libres, y finalmente, se presentan algunos ejemplos de sistemas de archivos típicos. El tercer capítulo expone las técnicas que se usan para administrar la memoria, haciendo énfasis en las más actuales, el paginado y la segmentación, pero también analiza técnicas más antiguas. La segunda sección presenta un caso de estudio que se dedica al sistema operativo Unix y sus descendientes. Esta sección se divide en dos capítulos: en el cuarto capítulo se estudian diversos aspectos acerca de esa familia de sistemas operativos, mientras que el quinto capítulo está dedicado a la programación en Shell script. El libro finaliza con unas conclusiones e incluye un anexo cuya función es explicar la instalación de máquinas virtuales.

Palabras clave: sistemas operativos, memoria, procesador, sistema de archivo, Unix, Shell script.

ABSTRACT

This book deals with the study of operating systems from the perspective of a career in the computer field. The book begins with an introduction that provides an overview of operating systems and then is divided into two sections. In the first section, the first chapter discusses how to schedule the central processing unit, and explains the different types of schedulers and the algorithms used to distribute processing time; problems of concurrent process synchronization are also analyzed. The second chapter is based on the abstract concept of file to explain how information is organized on external storage media; different ways of organizing space used by files and controlling free space are discussed; and finally, some examples of typical file systems are presented. The third chapter examines the techniques used to manage memory, focusing on the most recent ones, paging and segmentation, but also analyzes older techniques. The second section introduces a case study that revolves around the UNIX operating system and its descendants. This section is divided into two chapters: the fourth chapter examines various aspects of this family of operating systems, while the fifth chapter focuses on shell scripting. The book ends with some conclusions and includes an annex that explains the installation of virtual machines.

Keywords: operating systems, memory, processor, file system, UNIX, shell script.

¿Cómo citar este libro? / How to cite this book?

Lezcano-Brito, M. G. (2017). *Fundamentos de sistemas operativos.* Entornos de trabajo. Bogotá: Ediciones Universidad Cooperativa de Colombia. DOI: http://dx.doi.org/10.16925/9789587601077

AUTOR

MATEO G. LEZCANO BRITO

Graduado de Cibernética-Matemática en la Universidad Central "Marta Abreu" de Las Villas (UCLV), Santa Clara, Cuba (1982), Máster en Computación Aplicada (1995) y Doctor en Ciencias Técnicas (1998). Fue Profesor Titular de la UCLV en pregrado y posgrado de la UCLV. Ha sido profesor invitado en universidades cubanas y extranjeras (Yugoslavia, Perú, Colombia, Venezuela, Bolivia, México, Mozambique). Ha publicado 84 artículos científicos, siete libros de texto y ha participado en 120 eventos científicos. Es socio emérito y Premio Pablo Miquel de la Sociedad Cubana de Matemática y Computación (SCMC), de la cual ha sido vicepresidente nacional. Forma parte del Tribunal Permanente para otorgar el grado de doctor en Ciencias Matemáticas en Cuba. Ha impartido diversos cursos a nivel de pregrado y posgrado, y es miembro del claustro de las Maestrías en Ciencia de la Computación, Computación Aplicada de la UCLV y de la Maestría en Informática Aplicada a la Educación de la Universidad Cooperativa de Colombia (UCC).

AUTHOR

MATEO G. LEZCANO BRITO

Graduated in Cybernetics and Mathematics from Universidad Central "Marta Abreu" de Las Villas (UCLV), Santa Clara, Cuba (1982). With a Master's degree in Applied Computing (1995), and a PhD in Technic Sciences (1998). He was an Associate Professor at UCLV in undergraduate and postgraduate education and have been invited to universities in Cuba, Yugoslavia, Peru, Colombia, Venezuela, Bolivia, Mexico, and Mozambique. He have published eighty-four scientific articles and seven textbooks, he also have participated in 120 scientific events. He is an emeritus partner of the Cuban Society of Mathematics (SCMC), in which he have been the National Vice-president, and have received its Pablo Miquel Award. He is part of the permanent court which grants the PhD in Mathematics Science in Cuba. He imparted several courses in undergraduate and postgraduate education. He is a member of the Computing Science MA, of the Applied Computing MA in UCLV and of the Applied Computing in Education MA of the Universidad Cooperativa de Colombia (UCC).

Fundamentos de sistemas operativos

Entornos de trabajo

Mateo G. Lezcano Brito

Fundamentals of Operating Systems

Work Environments

Universidad Cooperativa de Colombia

EDICIONES
Universidad Cooperativa de Colombia

Catalogación en la publicación – Biblioteca Nacional de Colombia

Lezcano Brito, Mateo G.
Fundamentos de sistemas operativos: entornos de trabajo = Fundamentals of operating systems: work environments [recurso electrónico] / Mateo G. Lezcano Brito. -- Bogotá : Universidad Cooperativa de Colombia, 2018.
Recurso digital (1 archivo pdf: 6Mb). – (Colección general en docencia)

Incluye datos biográficos del autor. -- Contiene bibliografía.

ISBN 978-958-760-107-7 (digital)

1. Sistemas operacionales (Computadores) 2. Unix (Sistema operacional para computador) 3. Shell script (Programa para computador) I. Título II. Serie

CDD: 005.42 ed. 23
CO-BoBN– a1023196

Fundamentos de sistemas operativos. Entornos de trabajo

ISBN (digital): 978-958-760-107-7
DOI: http://dx.doi.org/10.16925/9789587601077

Colección General en Docencia

Proceso de arbitraje doble ciego:
Recepción: marzo del 2016
Evaluación propuesta de obra: abril del 2016
Evaluación de contenidos: agosto del 2016
Correcciones de autor: octubre del 2016
Aprobación: diciembre del 2016

Fondo Editorial

Director Nacional Editorial (E), Juan Pablo Mojica Gómez

Especialista en Gestión Editorial, Daniel Urquijo Molina

Especialista en Producción Editorial, Camilo Moncada Morales

Proceso editorial

Corrección de estilo, Daniela Echeverry

Lectura de pruebas, Hernando Sierra

Traducción al inglés, Nathalie Barrientos

Diseño y diagramación, Ivonne Carolina Cardozo Pachón

Diseño de portada, Ivonne Carolina Cardozo Pachón

Impresión, Xpress Estudio Gráfico ,

Impreso en Bogotá, Colombia. Depósito legal según el Decreto 460 de 1995.

CONTENIDO

CONTENTS

ÍNDICE DE FIGURAS

ÍNDICE DE FIGURAS

ÍNDICE DE TABLAS

Dedicatoria

Dedico este libro a la memoria de mis padres, ellos fueron la guía de mi vida y mi inspiración.

Mi padre, Mateo Lezcano Pérez, falleció hace ya algunos años y su vida fue un ejemplo de sacrificio en favor de nuestra familia. Siempre hablamos de él con alegría y nos da mucho gusto recordar sus ocurrencias y la forma persistente de buscar trabajo para mejorar la vida de la familia en una época muy dura de esa segunda madre que es mi Cuba querida.

Mi madre, Andrea Beatriz Brito Morales, falleció cuando estaba trabajando en los toques finales de este libro. Ella marcó la vida de toda la familia y gracias a su visión, sus cinco hijos se hicieron personas responsables que desean dar continuidad a su obra. Aún no puedo hablar de ella con alegría porque su fallecimiento está muy reciente y no logro adaptarme a la idea de no tenerla más, sé que lo lograré más adelante, para unir a ambos en ese recuento alegre que se hace de las personas que marcan nuestras vidas.

Nada de lo que diga podrá honrar lo suficiente a estos dos seres maravillosos, ellos escribieron este libro porque de ellos se deriva todo lo bueno que haya podido hacer y haré en la vida.

Gracias

Introducción

Debido a su complejidad, los sistemas operativos (SO) se diseñan e implementan por grupos de especialistas, y sus interioridades se estudian en carreras de perfil informático que tienen la necesidad de explotarlos lo más eficientemente posible o de hacer algunas modificaciones con el propósito de resolver problemas específicos no concebidos por los diseñadores originales.

Los SO también se pueden estudiar desde el punto de vista de los usuarios que no tienen necesidad de conocer sus interioridades para usarlos, lo cual es muy habitual hoy en día.

Este libro puede ser parte de un curso de SO dirigido a alguna carrera de perfil informático, aunque cualquier persona que esté interesada en estudiar las interioridades de los SO puede leerlo y comprenderlo, porque la intención es ofrecer explicaciones profundas pero al alcance de los lectores estudiosos.

Un SO actúa como un intermediario entre los usuarios de la computadora y el *hardware*. El objetivo es proporcionar un medio cómodo, eficiente y seguro para explotar los recursos de la computadora, sin tener que conocer los detalles específicos de cada uno de los componentes del sistema de cómputo. En este entorno, los usuarios pueden ser personas u otros programas.

Debido a su complejidad, los SO modernos deben diseñarse por partes o módulos que tienen responsabilidades específicas e interactúan entre sí para resolver las diversas tareas que tienen a su cargo.

ORIGEN DEL NOMBRE

En las primeras máquinas computadoras no se instalaba ningún programa para manipular los recursos, es decir, no existían en esa época los sistemas operativos. Una persona, conocida como **el operador**, muy especializada en una computadora en particular tenía el trabajo de operarla. Para que el operador pudiera realizar su trabajo, era necesario que conociera, en todos sus detalles, el *hardware* específico de la computadora con la cual actuaba.

El operador era el responsable de hacer muchas operaciones que hoy en día están automatizadas, entre ellas las siguientes:

* Iniciar la computadora, mediante un panel de control manual, que posteriormente se auxilió de un pequeño cargador sobre cinta de papel o tarjeta perforada.

* Cargar los programas que daban soporte a los demás programas, lo que hoy en día se conoce como **programas de sistemas**, por ejemplo: compiladores de diversos lenguajes, enlazadores, etc.
* Cargar los programas de usuarios y los datos asociados a ellos.
* Atender cualquier fallo que pudiera suceder y resolverlo, para lo cual a veces se hacía un vaciado de la memoria[1] con el objetivo de examinar los valores de los registros del procesador.

En la década de los cincuenta, se hicieron los primeros esfuerzos para automatizar los procesos descritos anteriormente. La idea de la automatización se basó en la observación de que el trabajo del **operador** de la computadora era bastante rutinario y seguía algunos principios que podían generalizarse.

La automatización de ese trabajo a través de un programa específico fue el embrión que dio origen a lo que más tarde se denominó **sistema de operación** en alusión al trabajo que hacía el operador.

El sistema operativo como parte del sistema de cómputo

Una computadora (o cualquier sistema que esté manipulado por un procesador) se puede ver como un conjunto de cuatro capas superpuestas (figura 1).

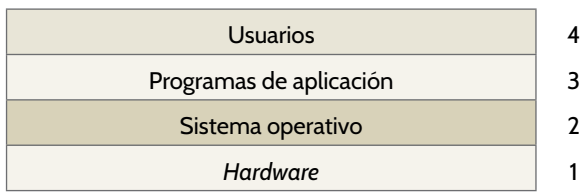


Figura 1. Sistema de cómputo estructurado por capas. Fuente: Elaboración propia.

* La capa inferior (capa 1) es el *hardware* y constituye la **máquina real** que se fabrica en una determinada empresa de construcción de computadoras.
* Cuando a la computadora o máquina real se le instala un SO (capa 2), se convierte en una máquina extendida que ofrece mejoras significativas a la computadora original. Las mejoras dependen del tipo de SO y de las potencialidades que brinda el *hardware*.
* En la tercera capa, se sitúan los programas de aplicación que hacen diversas labores a nombre de los usuarios y usan los servicios del SO para hacer su trabajo.
* En la cuarta capa, es decir, en la capa superior, se sitúan las solicitudes de los usuarios. Esas solicitudes se pueden hacer directamente o a través de los programas de aplicación.

1 Un vaciado de memoria es una copia del contenido de la memoria principal, en código de máquina (binario, octal o hexadecimal). Ese contenido se puede examinar para investigar las causas del funcionamiento anormal de un programa. Hoy en día casi no se usa, debido a la existencia de lenguajes de alto nivel y de programas de puesta a punto (*debuggers*) interactivos muy potentes.

Es importante recalcar que el *hardware* es muy diverso y por ese motivo no resulta fácil lidiar con cada una de sus especificidades; de hecho, en los primeros años de la computación solo personas muy especializadas se atrevían a enfrentarse a las computadoras, y por eso existía una separación entre los operadores de las computadoras y los programadores.

Quizás el lector que esté consultando este libro no lo pueda comprender pero un operador de computadoras era una persona muy especializada en un tipo de computadora específica y para operar otra debía entrenarse un buen tiempo en ella. Esta especialización se debía a que las computadoras se fabricaban con *hardware* muy específicos y por eso la forma de operarla difería de una a otra (aunque siempre existieron algunos principios generales).

Por el momento, se puede definir un SO como un programa que maneja los recursos de una computadora. Los recursos pueden ser de *hardware* o de *software*:

* Recursos de *hardware*: el procesador central (Central Process Unit - CPU); la memoria; los equipos de entrada/salida, tales como: el ratón (*mouse*), las impresoras, el teclado y el monitor, entre otros; los equipos de almacenamiento externo, por ejemplo: discos, memorias USB, cintas magnéticas, etc.
* Recursos de *software*: estructuras de datos diversas, compiladores, enlazadores, el intérprete de comandos del SO (Shell), los archivos y directorios, etc.

El SO también es un programa o, más bien, un conjunto de programas que conforman un sistema. Los programas de computadoras se pueden dividir en dos categorías generales:

* Programas de sistema. Son programas que dan apoyo a otros programas, por ejemplo: los compiladores, los editores de textos, los enlazadores, etc. El SO es el más importante y complejo de todos los programas de sistemas.
* Programas de aplicación. Son programas que tienen el objetivo de resolver algún problema específico, por ejemplo: un programa para hacer las nóminas de pago o para llevar la contabilidad de una empresa, un programa para controlar las historias clínicas de un hospital, etc.

El SO (el programa de sistema más importante) tiene la responsabilidad de administrar los recursos con que cuenta la computadora.

Estructura de los sistemas operativos

Al igual que la visión general que se tiene de un sistema de cómputo (figura 1), estructurado por partes que interactúan entre sí, los SO se organizan en módulos que asumen diferentes responsabilidades.

Se puede tomar como ejemplo básico al SO MS-DOS[2], el cual fue un sistema operativo que se diseñó sin tomar en cuenta la dimensión, más bien la popularidad, que iba

2 Microsoft Disk Operating System (MS-DOS). Diseñado por la compañía Microsoft en 1980, se basó en el SO QDOS de la compañía Seattle Computer Products. La versión para IBM se llamó PC-DOS: http://www.microsoft.com/

a alcanzar; por ese motivo no está bien estructurado en módulos con funciones específicas. En justicia, este SO estaba limitado por las propias fronteras que imponían los procesadores Intel 8086 e Intel 8088[3] sobre los cuales se implementó.

Los procesadores Intel 8086/88 solo tenían un modo de operación y los programas podían acceder a cualquier lugar de la memoria de un megabyte que estaba accesible desde el MS-DOS. Debido a esto, un programa mal intencionado podía escribir en el área del SO (a pesar de que el SO residía en un área específica de la memoria).

La versión original del SO UNIX[4, 5] también tenía una estructura dividida en dos partes, que estaba limitada por las propias restricciones del *hardware* de la época, pero en este caso se podía apoyar en procesadores con dos modos de operación (modo usuario y modo protegido).

UNIX estaba formado por dos módulos separados: el núcleo, o *kernel*, y los programas de sistemas. El núcleo era una estructura monolítica, es decir que constituía una sola unidad que estaba dividida internamente en varias funcionalidades, pero que funcionaba como un todo y por eso era muy difícil de mantener. Se debe agregar que esas partes eran demasiados grandes y abarcaban muchas funciones que deberían haber estado separadas.

La figura 2 muestra una idea esquematizada de la estructura de un SO monolítico. El núcleo siempre actúa en modo protegido mientras el resto del sistema actúa en modo usuario.

La modificación de cualquier componente de un núcleo monolítico implica que sea necesario compilar el núcleo por completo. Por ejemplo, si se modifica el subsistema de memoria del SO que se esquematiza en la figura 2, habrá que recompilar todo el núcleo, a pesar de que en este caso no se han modificado los restantes dos subsistemas que conforman el núcleo. Esta acción de recompilar puede consumir tiempo y también causar errores, en el peor de los casos.

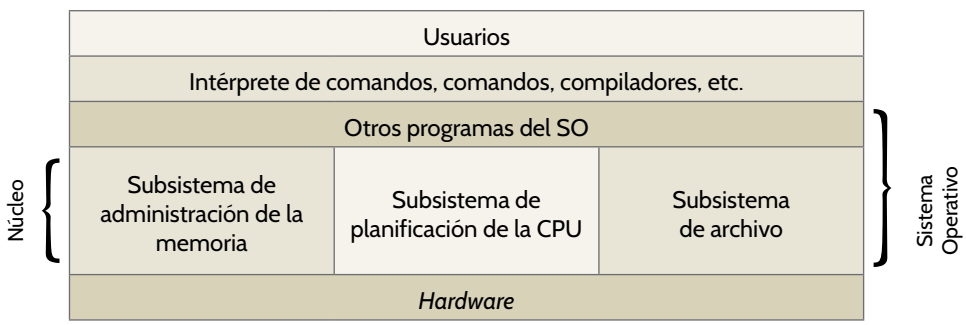


Figura 2. Estructura monolítica de un SO hipotético. Fuente: Elaboración propia.

Debe observarse que en esa estructura (figura 2), los módulos **administración de la memoria, planificación de la** CPU y de **archivos**, aunque tienen una cierta separación, están reunidos en un núcleo monolítico que actúa en modo protegido, los demás componentes

3 Primeros microprocesadores de 16 bits diseñados por la compañía Intel: http://www.intel.com/content/www/us/en/homepage.html

4 Desarrollado por Dennis Ritchie (fallecido en 2011) y Ken Thompson (Premios Turing 1983).

5 UNIX es la marca oficial de este SO, pero también se usa Unix, esta última sobre todo para referirse a los SO tipo Unix (*like Unix*).

del SO actúan en modo usuario. El intérprete de comandos (Shell) y los comandos se distribuyen junto al SO (aunque no forman parte de él) y contienen muchas funcionalidades adicionales muy importantes.

Una solución al problema del núcleo monolítico es migrar algunas de sus funcionalidades hacia capas superiores, de modo que se tenga un núcleo pequeño o **micro núcleo** y se formen capas superiores, con diferentes funcionalidades, que actúen de forma independiente (figura 3). El SO Minix[6], que se concibió con fines docentes, tiene esa arquitectura.

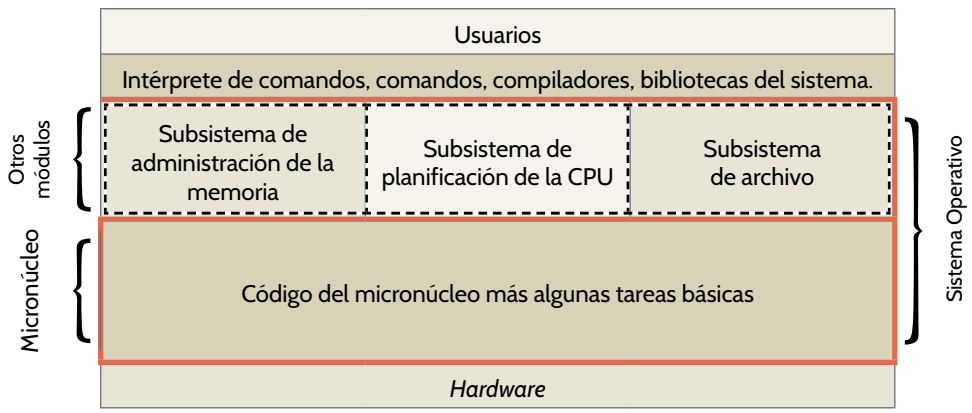


Figura 3. Estructura de micronúcleo de un SO hipotético. Fuente: Elaboración propia.

En el caso de la figura 3, algunas de las funcionalidades que estaban al nivel de otros módulos (posiblemente mal situadas) migran hacia el **micronúcleo**, el cual tiene una función básica, a la vez que sirve de medio de comunicación entre las partes que forman la capa superior. Obsérvense las líneas de punto discontinuas que separan los subsistemas del área identificada como "Otros módulos" para significar que son módulos independientes y que la modificación de uno no afecta a los demás. En el caso de la figura 3, el micronúcleo y la capa superior a él actúan en modo protegido, mientras que los restantes elementos de *software* actúan en modo usuario.

Una idea más actual es disponer de un núcleo que contiene un conjunto de componentes básicos y enlaces hacia otros módulos de servicios, que se cargan de forma individual cuando se necesiten. La carga de los módulos puede hacerse en el momento de iniciar el SO o en tiempo de ejecución solo cuando sea necesario.

El diseño del SO en módulos independientes hace más fácil la modificación de cada uno de los componentes, debido a que no hay necesidad de recompilar todo cuando se hace una modificación a alguna de sus partes. También es más fácil agregar nuevas funcionalidades al sistema, para lo cual solo hay que programar nuevos módulos y agregarle los enlaces necesarios al núcleo.

El diseño del SO en módulos relativamente independientes es muy común en las versiones actuales de Unix, por ejemplo Solaris, Linux y Mac OS[7], y también en las ediciones modernas de los SO de la compañía Microsoft.

6 Diseñado por Andrews Tanenbaum, se distribuye junto con su libro *Operating system. Design and implementation.*

7 Mac OS, Macintosh Operating System.

TIPOS DE SISTEMAS OPERATIVOS

Antes de especificar las diferentes clasificaciones, es importante entender el concepto de **proceso**. Ese concepto se detallará en el capítulo I, pero dada su importancia es necesario tener una idea acerca de él desde muy temprano, y distinguir claramente entre programa y proceso.

Un programa es un conjunto de órdenes escritas en algún lenguaje de programación, y en definitiva es un algoritmo que se ha programado usando el lenguaje seleccionado (C, Pascal, Java, Ensamblador, Prolog, Shell script, etc.).

El programa en sí mismo es un **ente pasivo**, pues desde que se escribe (ya sea en una hoja, en una pizarra, etc.) o se guarda en una unidad de almacenamiento, existe como tal pero no realiza ninguna tarea.

Por otra parte, un proceso se define como un programa en ejecución. Debe observarse que en este caso se habla de un **ente activo** que está realizando algunas tareas y para llevarlas a cabo necesita varios recursos, entre ellos, el procesador, la memoria, etc. Antiguamente, en lugar de la palabra "proceso" se usaba "tarea" (*task*), sobre todo porque se asociaba a los SO de tratamiento por lotes, los cuales manipulaban diversas tareas.

Una vez que se ha destacado la concepción de los procesos y su posición medular dentro del SO, se pueden comprender mejor algunas de las clasificaciones que se presentan a continuación y que toman en cuenta diversos puntos de vista que se analizarán luego.

Uno de los aspectos más importantes de los SO actuales es la capacidad de ejecutar varios procesos "a la vez", lo que se conoce con el nombre de multiprogramación o multitarea.

Para que un programa se pueda ejecutar, es necesario que esté cargado en memoria; una posible solución es la que se muestra en la figura 4, en la cual cuatro procesos y el SO comparten una memoria de longitud M. Como se verá más adelante, no es necesario que el proceso completo esté cargado en la memoria (aunque sí una parte de él). Esa idea (de no tener el proceso completamente cargado en memoria) permite hacer creer que la memoria es mayor de lo que es realmente (también se verá más adelante).

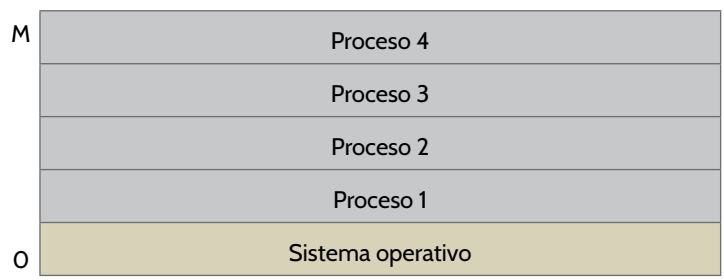


Figura 4. Varios procesos en memoria permiten multiplexar[8] el procesador. Fuente: Elaboración propia.

Con base en el análisis anterior, es decir, de acuerdo con la cantidad de procesos que se pueden atender, los SO pueden clasificarse como:

8 Multiplexar significa combinar fuentes independientes de información (el término proviene de las telecomunicaciones). Para este caso, es intercambiar el uso de la CPU entre diferentes procesos.

* Multiprogramados o multitarea (*multi-programming, multi-tasking*). Permiten ejecutar varios procesos "a la vez". Ejemplos: Windows, Unix.

* Monoprogramados o monotarea (*mono-programming, mono-tasking*). Solamente permiten que se esté ejecutando un proceso, el cual es dueño de toda la memoria de la computadora (figura 5). Ejemplos: MS-DOS, CP/M[9].

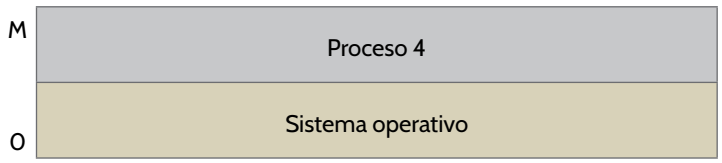


Figura 5. Situación de la memoria en un SO monoprogramado. Fuente: Elaboración propia.

La multiprogramación se logra compartiendo el procesador, es decir, en realidad solo un proceso se ejecuta en cada instante de tiempo. Por ese motivo, la expresión **a la vez** aparece entrecomillada en los párrafos anteriores, pues en realidad en cada instante de tiempo solo un proceso es dueño del procesador y está ejecutando una de sus instrucciones. El SO intercambia el procesador entre los procesos que tiene cargados en memoria dando la sensación de que todos ejecutan al mismo tiempo.

Hoy en día, es muy común que una computadora tenga más de un procesador, sobre cada uno de ellos se puede ejecutar un proceso diferente. Obsérvese que en este caso esos procesos sí se estarían ejecutando a la vez.

Los SO que permiten el uso simultáneo de más de un procesador se denominan de **multiprocesamiento**.

Con base en este análisis, es decir, de acuerdo con la cantidad de procesadores que pueden atender, los SO pueden ser:

* De multiprocesamiento (*multiprocessing*), que permiten explotar varios procesadores. Ejemplos: Windows (8, 10), Ubuntu, Debian, etc.

* De monoprocesamiento (*single processing*), que solamente pueden explotar un procesador, aunque la computadora sobre la que estén instalados tenga más de uno. Ejemplos: Windows 95, Windows 98, etc.

Otra clasificación toma en cuenta la cantidad de usuarios que pueden estar conectados al SO, realizando diversas tareas. Desde ese punto de vista, los SO pueden ser:

* Monousuario (*single user*), solamente un usuario puede conectarse al sistema. Ejemplos: MS-DOS, CP/M.

* Multiusuario (*multi-user*), varios usuarios pueden estar conectados y trabajando con el SO en el mismo momento, por ejemplo, en Unix y Linux distintos usuarios se conectan al mismo SO desde diversas **terminales**.

9 Control Program for Microcomputers (CPM), sistema operativo de la década de los setenta.

Por último, existe una clasificación que toma en cuenta el tiempo de respuesta o la forma de interactuar con las tareas. En ese sentido, los SO se clasifican como:

* De tiempo real (*real time*). Son SO que deben dar respuestas rápidas, inmediatas o casi inmediatas. Ejemplos: QNX[10], RTLINUX[11].
* De tratamiento por lotes (*batch*). Estos SO se caracterizan por realizar grandes lotes de trabajos con características comunes, que no exigen interacción con los usuarios y que proporcionan las respuestas al final de la ejecución de todo el lote. Hoy en día resulta muy difícil encontrar "un sistema por lotes puro" y muchos SO lo que hacen es incluir facilidades para ejecutar tareas en lotes. El ejemplo más común en este sentido es el uso de los archivos **.bat**[12] de la familia de SO Windows (y su antecesor MS-DOS), que pueden contener secuencia de comandos para ejecutarse en lotes controlados por medio del lenguaje *script*, que forma parte del intérprete de comandos de esos SO.

Debe observarse que un SO se puede incluir dentro de varias categorías; por ejemplo, el SO Linux es de multiprocesamiento, multiprogramado y multiusuario.

EL INTÉRPRETE DE COMANDOS O SHELL

El Shell es un componente que usualmente se distribuye junto con el SO, aunque no forma parte de él. Se encarga de leer las órdenes que teclean los usuarios para analizar si están correctas, **interpretarlas** y ordenar su ejecución. Se denomina Shell porque, al igual que la **concha** de un molusco, rodea al SO aislando a los usuarios de las interioridades del SO. Los intérpretes de comandos pueden ser de texto o gráficos.

En los SO de la familia Unix, existen varios Shell; por ejemplo, bash, ash, csh, Zsh, ksh, tcsh, C *shell*, etc. Los usuarios pueden elegir el Shell que deseen, el capítulo V versa acerca de la programación en uno de ellos, el **bash**.

FORMA EN QUE LOS SISTEMAS OPERATIVOS BRINDAN SUS SERVICIOS

Como se ha mencionado, no había SO para las primeras computadoras y por eso cada programa que se ejecutaba tenía que especificar órdenes detalladas para que funcionaran los distintos tipos de *hardware*. Esa era una tarea muy engorrosa que, afortunadamente, ya no hay que hacer porque el SO, con la ayuda de los manipuladores específicos de algunos equipos, se encarga de realizar la labor.

Los SO brindan sus servicios a otros programas y usuarios a través de **programas de sistemas** y **llamadas al sistema.**

PROGRAMAS DE SISTEMAS

Los programas de sistemas que acompañan al SO tienen una funcionalidad específica, se conocen también como **comandos** y pueden ser internos o externos.

10 Se usa principalmente en sistemas embebidos. Es un SO de micronúcleo, tipo Unix.

11 Ejecuta Linux como un hilo de ejecución (*thread*) de menos prioridad que las tareas de tiempo real.

12 Se puede analizar un conjunto de buenos ejemplos en: http://www.robvanderwoude.com/batexamples.php

Los comandos internos se denominan así porque forman parte del código del intérprete de comandos (Shell) y se cargan en la memoria junto con él. Algunos ejemplos de comandos internos de los SO de la compañía Microsoft son: dir, type, copy; mientras pueden mencionarse, entre otros, los comandos bg, cd, fg en los SO de la familia Unix. Como es lógico, en una máquina que tenga instalado alguno de esos SO no existe ningún archivo de programa con esos nombres, debido a que ellos son parte del Shell.

Los comandos externos son programas independientes que vienen junto con el SO y que normalmente se guardan en alguna parte del disco durante el proceso de instalación. Por ejemplo, en los SO de la familia Windows esos comandos están en el directorio \Windows\System32 del disco donde esté instalado el SO; algunos ejemplos son: chkdsk, at y comp. Algunos de los comandos externos de Unix son: find, cp, help, que se localizan en los directorios /bin o /usr/bin.

LLAMADAS AL SISTEMA

Una llamada al sistema es una invocación a una función que reside en el núcleo del SO, lo cual quiere decir que cuando un proceso cualquiera quiere hacer una acción que solo la puede hacer el SO, tendrá que hacerla a través de una llamada al sistema.

Algunas llamadas al sistema de los SO Windows[13] son: NtSetEvent(), NtSetInformationToken(), etc. Como ejemplos de llamadas al sistema en los SO de la familia Unix[14], se pueden citar las siguientes: creat(), open(), exec(), etc.

LOS SISTEMAS OPERATIVOS, LA OPERACIÓN DUAL Y LA PROTECCIÓN

Como se ha mencionado, el SO es un programa compuesto por diferentes partes que pueden actuar de manera más o menos independiente. Esas partes o módulos pueden crear diversos procesos con responsabilidades específicas que necesitan determinadas prioridades.

La mayoría de los procesadores modernos permiten dos modos de operación:

* Modo usuario o real.
* Modo protegido (también se conoce como modo sistema, modo supervisor y modo núcleo o *kernel*).

Cuando se enciende una computadora, el procesador arranca en **modo usuario** y el SO cambia después el procesador a **modo protegido**. Los procesos de usuario actúan siempre en modo usuario, y la mayoría de los procesos del SO actúa en modo protegido. La capacidad del SO de cambiar de un modo a otro se conoce como **operación dual**. Cuando un proceso de usuario necesita hacer alguna operación que solo puede realizar el SO, se ejecuta una serie de pasos que se describen a continuación (en forma general) y que se aprecian en la figura 6:

13 Puede consultarse el siguiente sitio: http://j00ru.vexillium.org/ntapi/

14 Consúltese el siguiente sitio: http://www.di.uevora.pt/~lmr/syscalls.html

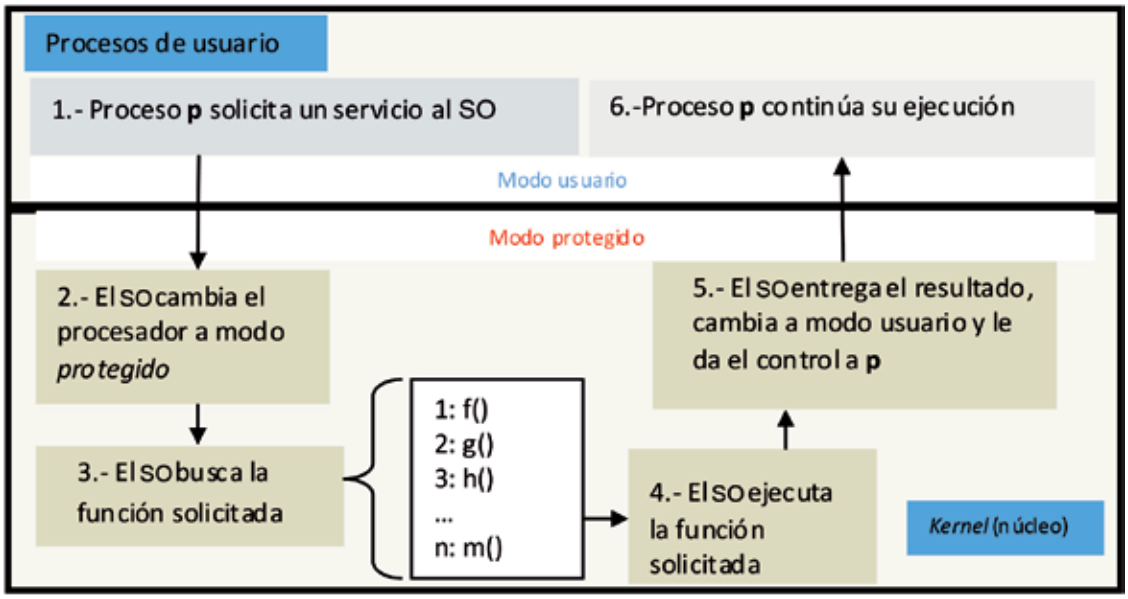


Figura 6. Transición entre modos de operación. Fuente: Elaboración propia.

1. El proceso (**p** en este caso), que se está ejecutando en **modo usuario**, le hace una **solicitud de servicio** al SO. Esa solicitud se conoce como una **llamada al sistema**, debido a que se está llamando al SO para que haga algo a nombre del proceso. Las funciones que responden a las llamadas al sistema están en el núcleo del SO y actúan en modo protegido.
2. El SO **cambia el procesador** de modo usuario a modo protegido (solo él puede hacer esa acción).
3. El SO busca, en un lugar específico de su núcleo, una tabla de saltos o de direcciones (simbolizada por 1:, 2:, 3:, …, n:; en la figura 6) que contiene las direcciones a las funciones que deben realizar el servicio solicitado, es decir, f():, g():, h():, …, m():, en este caso (figura 6). Por ejemplo, si el salto se produce a la dirección 3:, la acción que se efectuará es la contenida en la función h().
4. Una vez encontrada la función solicitada, el SO la ejecuta.
5. El SO envía los resultados al proceso solicitante, cambia el procesador a modo usuario y le da el control al proceso que solicitó el servicio (**p** en el ejemplo).
6. El proceso (**p** en este caso) continúa su ejecución.

Esta forma de proceder es una manera apropiada para proteger los recursos del sistema. Los procesadores Intel 8086/8088 sobre los que se implementó el SO MS-DOS no tenían esa capacidad, de ahí que el SO no podía hacer gran cosa para protegerse sin contar con el apoyo del *hardware*.

HIPERVISOR O MONITOR DE MÁQUINAS VIRTUAL

Debido a la popularidad que han alcanzado las máquinas virtuales[15] hoy en día y a las posibilidades que ofrecen para hacer estudios de los SO, se hace un paréntesis para explicar estos conceptos.

Un **hipervisor**, o monitor de máquina virtual (*virtual machine monitor*- VMM), es un *software*, *firmware* o *hardware* que crea y ejecuta máquinas virtuales. La computadora sobre la que el hipervisor se ejecuta se denomina máquina anfitriona (*host*), y se le llama máquina cliente (*guest*) a cada una de las máquinas virtuales que se ejecutan sobre la máquina hospedera.

La idea, en este caso, es abstraer el *hardware* de la computadora anfitriona (todos sus componentes) sobre diferentes ambientes de ejecución para crear la ilusión de que cada ambiente se ejecuta sobre una computadora particular.

La máquina virtual no ofrece nuevas funcionalidades, sino que presenta una interfaz idéntica a la máquina simple que está debajo de ella, emulando[16] todo su *hardware* (aunque con limitaciones de capacidad, por ejemplo, menos memoria). Es importante que no se confunda emular con simular[17].

La capa de funcionalidades que se le agrega al *hardware* o al SO anfitrión es el VMM o hipervisor, y presenta una plataforma operativa virtual (***hardware* virtual**) a los SO invitados, a la vez que monitorea su ejecución.

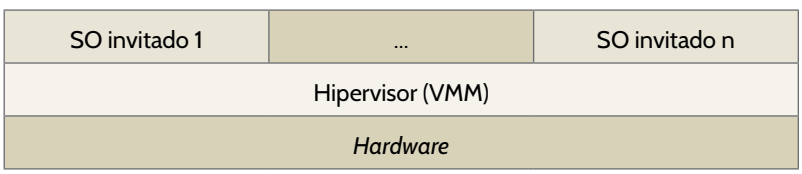


Figura 7. Concepción de VMM directamente sobre el *hardware*. Fuente: Elaboración propia.

Existen dos tipos de hipervisores o VMM (algunos autores exponen una tercera categoría, aunque incluso estas dos muchas veces son discutidas debido a varios factores).

* Los hipervisores de tipo 1 (figura 7) se ejecutan directamente sobre el *hardware* físico. Este tipo de hipervisor controla todos los accesos al *hardware* y es la forma más antigua de virtualización, pero también es la más potente, y por eso se usa mucho actualmente. Son ejemplos de este tipo de VMM: Microsoft Hyper-V, Citrix XEN Server y VMWare ESX-Server.

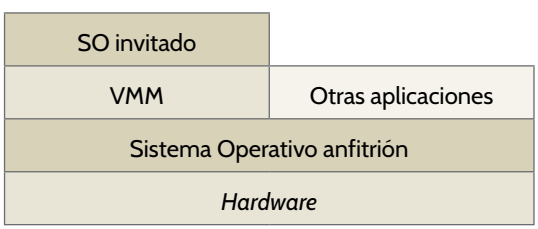


Figura 8. Concepción de VMM sobre un SO Fuente: Elaboración propia.

15 Virtual: algo que se comporta como real, a pesar de no serlo.

16 Emular: modelar de forma precisa algún dispositivo. En este caso, una computadora con todo su *hardware*.

17 Simular es reproducir un comportamiento de algo, haciendo que parezca real.

* Los hipervisores de tipo 2, no se instalan directamente sobre el *hardware*, sino sobre un SO (denominado anfitrión), lo que agrega una capa más al sistema (figura 8). La máquina virtual de Java (JVM) tiene esa concepción y los sistemas VirtualBox, VMWare y otros son ejemplos que pertenecen a este tipo de VMM.

Para realizar las prácticas que se proponen en los cursos de SO, es conveniente usar máquinas virtuales. El anexo A muestra una guía para instalar SO sobre máquinas virtuales, donde se explica cada uno de los pasos que se dan durante ese proceso; el objetivo es que los estudiantes comprendan los pasos para instalar SO y sepan tomar decisiones adecuadas ante las preguntas de los asistentes de instalación.

ORGANIZACIÓN DEL RESTO DE LOS CONTENIDOS

Hasta este punto, se ha pretendido dar una visión general de los SO para entender cada uno de los capítulos que siguen. Muchos de los aspectos que se han presentado en esta breve introducción se retomarán más adelante.

El libro se divide en dos secciones, una general y otra específica.

La primera sección abarca los capítulos I, II y III. En ellos se estudian las diferentes tareas que debe llevar a cabo un SO, las cuales están integradas en partes o módulos del SO y a cada una se dedicará un capítulo:

* El capítulo I, "Manipulación de procesos e hilos", trata acerca de las distintas formas que existen para planificar el uso del procesador central. Explica los diferentes tipos de planificadores y los algoritmos que se usan para distribuir el tiempo de procesamiento. También se analizan los problemas de sincronización entre procesos concurrentes.
* El capítulo II, "El sistema de archivo", versa acerca de la forma en que se organiza la información sobre los soportes de almacenamiento externos, para lo cual se basa en el concepto abstracto de archivo. Se discuten las distintas maneras de organizar el espacio ocupado por los archivos y el control sobre los espacios libres. Finalmente, se presentan algunos ejemplos de sistemas de archivos típicos.
* El capítulo III, "Administración de la memoria", finaliza la primera sección y expone las técnicas que se usan para administrar la memoria, con énfasis en las más actuales, el paginado y la segmentación, pero también explicando las más antiguas, de las cuales se han tomado algunas ideas que aún son válidas.

La segunda sección, dedicada al estudio del SO Unix y sus descendientes, se divide en dos capítulos:

* En el capítulo IV, "Generalidades del SO Unix y sus descendientes", se estudian diversos aspectos acerca de esa familia de SO, entre los se incluyen: los Shell, el trabajo con procesos, las tuberías y los filtros.
* El capítulo V está dedicado a la programación en Shell script.

BIBLIOGRAFÍA CONSULTADA

Abraham, S., Baer- Galvin, P., & Gagne, G. (2013). *Operating system concepts* (9.ª ed.). Nueva York: John Wiley & Sons.

Elmasri, R., Carrick, A., & Levine, D. (2009). *Operating systems: A spiral approach.* Nueva York: McGraw-Hill.

Harvey, M., Deitel, P., & Choffnes, D. (2003). *Operating Systems* (3.ª ed.). Nueva Jersey: Prentice Hall.

Stallings, W. (2014). *Operating systems: Internals and design principles* (8.ª ed.). Londres: Pearson.

Tanenbaum , A. (2006). *Operating systems: Design and implementation* (3.ª ed.). Nueva Jersey: Prentice Hall.

Tanenbaum, A., & Bos, H. (2014). *Modern operating systems* (4.ª ed.). Londres: Pearson.

Carretero, J., de Miguel, P., García, F., & Pérez, F. (2007). *Sistemas operativos. Una vision aplicada* (2ª ed.). Nueva York: McGraw-Hill.

CAPÍTULO I

Manipulación de procesos e hilos

RESUMEN

Este capítulo aborda la utilización del procesador central y las políticas para planificar su uso, describiendo el bloque de control de proceso como estructura de datos básica para preservar los atributos asociados a los procesos, con el propósito de planificarlos, detenerlos y asignarle el procesador en ambientes multiprogramados. Se detallan los intervalos de vida de los procesos, así como sus estados. El capítulo explica los algoritmos de planificación de procesos, los distintos planificadores y la forma como interactúan entre ellos, así como la relación entre el planificador de periodo corto y el despachador, detallando los objetivos de la planificación de procesos (ligeros y pesados); además, hace un análisis acerca de las operaciones que se pueden efectuar sobre los procesos y la forma como dichas entidades se comunican entre sí. También se ofrecen detalles acerca de las diferencias que existen entre los procesos pesados y los procesos ligeros o hilos, ofreciendo una panorámica acerca de los beneficios de usar los hilos, a la vez que explica algunos elementos esenciales de las bibliotecas de hilos Pthreads y Win32. Se analizan los problemas de concurrencia, el problema de la sección crítica y las distintas estrategias que se pueden utilizar para sincronizar la ejecución de procesos concurrentes, desde la perspectiva de soluciones apoyadas por *software* o por *hardware*. El capítulo finaliza con un resumen y una sección de ejercicios propuestos.

Palabras clave: procesador, procesos, multiprogramación, concurrencia, sección crítica, planificación de procesos.

¿Cómo citar este capítulo? / How to cite this chapter?

Lezcano-Brito, M. G. (2017). Manipulación de procesos e hilos. En *Fundamentos de sistemas operativos. Entornos de trabajo* (pp. 27-76). Bogotá: Ediciones Universidad Cooperativa de Colombia.

CHAPTER I

Process manipulation and threads

ABSTRACT

This chapter discusses the use of the central processing unit and the policies to schedule its use, describing the process control block as a basic data structure to preserve the attributes associated with processes in order to schedule them, stop them and assign them a processor in multi-programmed environments. Life intervals of processes, as well as their states, are detailed. The chapter explains process scheduling algorithms, different schedulers and how they interact with each other, as well as the relationship between the short-term scheduler and the dispatcher, detailing the objectives of process (lightweight and heavyweight) scheduling; in addition, the operations that can be carried out on processes and how such entities communicate with each other are analyzed. Details on the differences between heavyweight processes and lightweight processes or threads are given, providing an overview of the benefits of using threads, while explaining some essential elements of Pthreads and Win32 thread libraries. Concurrency problems, critical section problems and various strategies that can be used to synchronize execution of concurrent processes are analyzed from the perspective of solutions supported by software or hardware. The chapter ends with a summary and a section of proposed exercises.

Keywords: Processor, processes, multiprogramming, concurrency, critical section, process scheduling.

I.1 INTRODUCCIÓN

Las primeras computadoras solo podían manipular un proceso a la vez, el cual tenía el control absoluto sobre todos los recursos del sistema de cómputo. Con el desarrollo vertiginoso del *hardware*, ese panorama ha cambiado radicalmente y hoy en día es posible cargar varios programas en memoria para que se ejecuten de manera concurrente.

La situación anterior ha obligado a establecer un control más estricto sobre los recursos dado que, al ser estos finitos y tener que compartirse entre varios procesos, surgen conflictos que pueden entorpecer la idea de realizar más tareas en el mismo tiempo físico.

A la definición inicial de proceso como un programa en ejecución, planteada en la introducción de este libro, se le puede agregar que es **una unidad de planificación y despacho**, de forma tal que una definición más precisa sería la siguiente: **un proceso es un programa en ejecución y una unidad de planificación y despacho.**

El hecho de que un proceso sea un **ente activo** queda implícito cuando se dice que el proceso es un **programa en ejecución**, es decir que realiza diversas acciones para las cuales necesita varios recursos (procesador, memoria, etc.). Por otra parte, el proceso es una **unidad de planificación** porque los planificadores deben decidir cuándo y a quién le entregan el procesador (a qué proceso, durante qué tiempo). Una vez tomada la decisión de a quién entregarle el procesador, será necesario **despachar el proceso** (entregarle el procesador).

La concepción del proceso como unidad de planificación y despacho es el objetivo central de este capítulo.

Para comprender los temas que se tratan, resulta de suma importancia entender el concepto de proceso en toda la magnitud de la palabra; nunca debe confundirse con el concepto de programa, dado que mientras el primero es un ente activo que posee recursos y tiene vida (realiza determinada actividad), el segundo es un ente pasivo que no hace absolutamente nada.

Quizás la diferencia entre proceso y programa señalada antes parezca una sutileza, pero un programa no deja de serlo por el hecho de estar escrito en un papel o en una pizarra, o de estar contenido dentro de un archivo que reside en cualquier medio de almacenamiento. En este último caso, existe la posibilidad de que el SO lo cargue en una parte de la memoria y que después, cuando pueda, le asigne el procesador; con estos dos recursos (la memoria y el procesador), el proceso puede comenzar a ejecutarse, aunque es posible que necesite algunos recursos adicionales.

Debe aclararse que el programa se convierte en proceso desde el momento en que el SO lo carga en memoria (parcial o totalmente) y construye algunas estructuras de datos para controlarlo. El proceso seguirá vivo hasta que finalice (normal o anormalmente), aunque por instantes de tiempos específicos no tenga asignado el procesador.

Posiblemente, el lector ha usado el administrador de tareas de los SO de la compañía Microsoft y habrá observado que, a pesar de no haber ordenado la ejecución de programa alguno, hay muchos procesos que se están ejecutando, ellos son parte de los procesos que inicia el SO. También se habrá percatado de que se pueden realizar "varias tareas a la vez", por ejemplo: editar un texto, imprimir un documento, navegar por Internet, escuchar música, todo eso lo podrá hacer aun cuando la máquina que esté usando tiene un solo procesador. ¿Cómo se logra eso?, en el resto del capítulo se abordan diferentes temas que darán respuesta a este interrogante.

I.2 EL BLOQUE DE CONTROL DE PROCESO

Como ya se ha dicho, un sistema multiprogramado permite que varios procesos estén en ejecución de forma simultánea, aunque la computadora sobre la cual realicen sus acciones tenga un solo procesador. Lo anterior implica que, de alguna forma, se deba compartir el procesador, debido a que solo se puede ejecutar una instrucción en cada instante de tiempo cuando el sistema de cómputo tiene un solo procesador.

Para detener un proceso temporalmente y después reiniciarlo por el mismo lugar que iba cuando fue detenido, es necesario guardar una instantánea de su estado. Con ese propósito, los SO usan una estructura de datos que se denomina Bloque de Control de Proceso (Process Control Block - PCB).

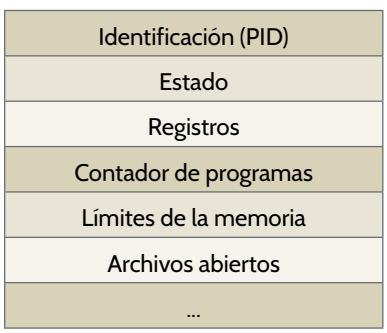


Figura I.1. PCB típico. Fuente: Elaboración propia.

La información que se debe guardar en el PCB es diversa y puede diferir de un SO a otro, pero algunos campos tienen que estar presentes de forma obligatoria, estos son (figura I.1):

* La **identificación del proceso**. Identifica unívocamente al proceso dentro del sistema, suele ser un número y se conoce muchas veces por sus siglas en inglés: PID (*process identification*).
* El **estado** del proceso. Es una palabra que describe lo que está haciendo el proceso en ese instante (esperando, ejecutando, bloqueado, etc.).

* El **contador de programa**. Contiene la dirección de la próxima instrucción a ejecutar. Ese valor se refiere al valor de un **registro** del procesador en un momento dado[18].
* Una copia de los valores de los restantes **registros del procesador**. Debe observarse que el campo "contador de programa" podría estar en este lugar (como un registro más), pero debido a su importancia, muchos SO lo sitúan en una posición especial dentro del PCB.
* Los **archivos abiertos**. Este campo puede ser un puntero a una tabla que contiene la lista de archivos abiertos por el proceso, en la cual se controlan cosas tales como: el lugar desde o hacia dónde se leerá o escribirá la próxima información, el tipo de acceso, etc. El sistema de archivos es el tema del capítulo II.
* Los **límites de memoria**. Indican la parte de memoria que tiene reservado el proceso y dependerá del esquema de asignación de memoria que use el SO. Este aspecto será tratado en el capítulo III.

La multiprogramación permite que varios procesos se ejecuten "a la vez", aun en máquinas que poseen un solo procesador. "A la vez" está entrecomillado porque en realidad el SO lo que hace es compartir el tiempo del procesador y dar la idea (los procesadores modernos son veloces) de que todo se hace al mismo tiempo. Para hacer eso, el SO debe interrumpir el proceso que se está ejecutando y marcar por dónde iba (tal como se hace cuando se está leyendo un libro y alguien o algo interrumpe).

La marca, en este caso, no puede ser física (como en el libro cuando se pone un marcador) y por eso los SO usan el PCB para guardar la información de los procesos interrumpidos, de modo que puedan reiniciarse cuando se les vuelva a asignar el procesador.

Un proceso necesita más que el código para ejecutarse, también necesita tener un control sobre sus recursos, tales como: su espacio de direcciones de memoria, la pila y los valores de los registros del procesador, entre otros, ese control se lleva a cabo por los diferentes módulos del SO, que basan su trabajo en la información contenida en el PCB.

I.3 INTERVALOS DE VIDA Y ESTADOS DE UN PROCESO

La vida de los procesos se caracteriza por transcurrir en dos intervalos de acciones que también se conocen como ráfagas (*burst*) o etapas:

* Ráfagas de CPU (CPU *burst*). Se refiere al tiempo en que el proceso es dueño del procesador, es decir, cuando está en el estado de ejecución.
* Ráfagas de entrada/salida (E/S o I/O *burst*). Se refiere al tiempo en que el proceso está esperando por la transferencia de datos (de entrada o de salida).

Cuando muchos procesos están activos dentro de un sistema multiprogramado (multitarea), se puede afirmar que cada uno tiene su **CPU virtual**, lo que significa que cada

18 Los procesadores poseen un **registro contador de programa**, donde se almacena la dirección de la próxima instrucción a ejecutar; por ejemplo, EIP en los procesadores Intel y RIP en los procesadores AMD.

proceso tiene una especie de copia de la CPU. Esa copia se guarda en el Bloque de Control de Proceso de cada proceso.

La idea básica para detener procesos e intercambiar las tareas, de modo que teniendo una sola CPU parezca que se ejecutan varias tareas a la vez, se basa en los dos siguientes aspectos:

- Cada vez que un proceso sea detenido, por cualquier motivo (se le terminó el tiempo de CPU, hizo una petición de E/S, etc.), los valores de la CPU se guardan en su PCB (la CPU virtual de ese proceso).
- Cada vez que un proceso vaya a entrar en estado de ejecución, se toman los valores de su PCB (su CPU virtual) y se copian en los registros correspondientes del procesador de la computadora, con lo cual la CPU queda tal como estaba cuando el proceso que se va ejecutar fue interrumpido en el pasado, es decir:
  - El valor adecuado del registro contador de programa, a fin de saber cuál es **su próxima instrucción a ejecutar**.
  - Los valores de los registros asociados a las direcciones de memoria del proceso, para saber **en qué parte de la memoria** están sus instrucciones y datos.
  - Los valores de los registros asociados a las tablas de archivos abiertos **para continuar leyendo o escribiendo** desde o hacia ellos, etc.

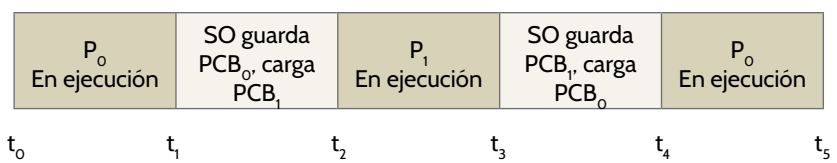


Figura I.2. Cambio de CPU entre procesos. Fuente: Elaboración propia.

Observe la figura I.2:

1. En el instante t_0, el proceso P_0 comienza su ejecución que continúa hasta el instante t_1; inmediatamente después del instante t_1, y por algún motivo que no interesa en este caso, el SO le quita el procesador al proceso P_0.
2. El SO guarda, en el PCB del proceso P_0 (PCB_0), todos los valores analizados anteriormente (quizás algunos más, en dependencia del SO).
3. Después, el SO decide que le va a dar el control al proceso P_1, para lo cual toma los valores de la CPU virtual del proceso P_1 que están guardados en su PCB (PCB_1) y carga con ellos la CPU de la computadora. El SO ha tardado el tiempo que hay desde el instante $t_1 + \Delta_{ti}$ en que tomó el control hasta el instante $t_2 - \Delta_{tj}$, y todos los procesos, excepto el SO, han estado detenidos durante ese tiempo.

4. Después, el SO le da el control al proceso P_1, el cual ejecuta hasta el instante t_3 y de nuevo el SO entra en acción para hacer el **cambio de contexto**, que no es más que la acción que toma el SO para garantizar la ejecución futura de P_1 a partir del momento en que fue interrumpido, es decir que repite los pasos del 1 al 3, pero ahora guarda los valores de P_1 y carga los de P_0.

En el ejemplo de la figura I.2, se han usado dos procesos (para hacer más simple la explicación), pero podrían ser muchos.

En un sistema multiprogramado, no se puede programar haciendo suposiciones acerca de la velocidad de ejecución, pues no todos los procesos realizan sus acciones a igual velocidad, e incluso, en el caso de que se ejecuten varias veces, cada vez no tienen que comportarse igual. Por ejemplo, en la situación anterior los procesos tuvieron que estar inactivos en varios intervalos de tiempo, porque el SO estaba haciendo su labor de cambio de contexto o porque el otro proceso era el que poseía el procesador, pero esos detalles pueden variar de una situación a otra y dependen de varios factores.

La vida de un proceso pasa por diferentes estados que describen, de cierta forma, la actividad que están realizando en cada instante de tiempo. En general, esos estados pueden ser los siguientes (los nombres pueden variar de un SO a otro):

* Nuevo (*new*). Un proceso está en el estado nuevo cuando se crea, es el momento en que ocurre la **transición de programa a proceso**. En ese estado nunca ha ejecutado, solo se le ha creado y actualizado el PCB dándole una identificación y se le ha asignado memoria, entre otras cosas.

* Ejecutando (*running*). Un proceso que **tiene el procesador** asignado.

* Listo (*ready*). Describe el estado de un proceso que está **listo para ejecutar** y solo espera que le asignen la CPU para hacerlo. Se llega a este estado bajo las siguientes condiciones:

  - Un proceso en el estado nuevo se admite en la cola de procesos que compiten por usar el procesador.

  - Un proceso que estaba haciendo un evento externo al procesador ya terminó y necesita de nuevo el procesador.

  - A un proceso que está en estado de ejecución se le quita el procesador (se dice que se desaloja o interrumpe) antes de terminar la etapa actual de ejecución.

* En espera (*waiting*). Un proceso que está **en espera de que finalice algún evento** externo (por ejemplo, una secuencia de lectura-escritura del disco).

* Terminado (*finished*). El proceso ya **terminó** todas sus acciones y queda en este estado hasta dejar de existir. El SO necesita un tiempo para liberar algunos recursos, por ejemplo el PCB.

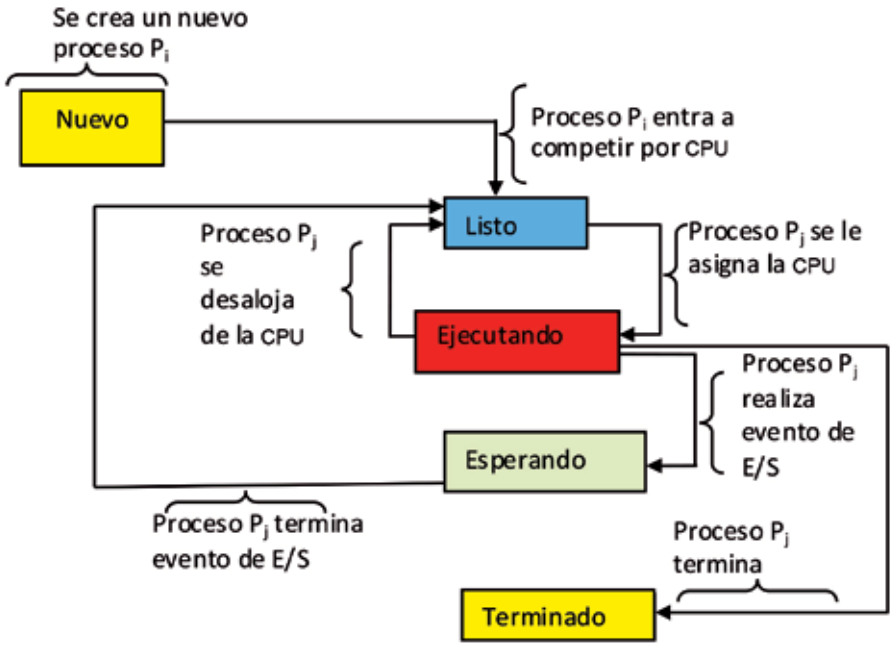


Figura I.3. Transición de estados de un proceso ($i \neq j$). Fuente: Elaboración propia.

La figura I.3 muestra los cambios de estados de los procesos. Obsérvese que se especifica $i \neq j$, para hacer notar que un proceso P_i que entra a competir por el procesador (pasa de **nuevo a listo**, de **corriendo a listo** o de **esperando a listo**) se ubica en una cola de procesos listos, y la única forma de entrar directamente a usar el procesador es que dicha cola esté vacía. Cuando un proceso P_j cualquiera (incluyendo el caso cuando $i = j$) posee el procesador, puede ser desalojado por las políticas de planificación que estén definidas, y en ese caso, se incorpora a la cola de listos debido a que el proceso sigue necesitando el procesador.

Cuando un proceso P_j necesita hacer un evento de entrada/salida, pasa al estado de esperando hasta que termine de hacer esas acciones con algún equipo, después de lo cual, ya con los datos que necesita, se incorpora a la cola de listos para competir de nuevo por el procesador.

Por último, cuando un proceso P_j finalice de hacer su trabajo, pasa al estado de terminado, desde el cual —y después de una breve transición— dejará de existir.

I.4 PLANIFICACIÓN DE PROCESOS

Como ya se ha dicho, el objetivo de la multiprogramación es tener varios procesos ejecutando "a la vez". El propósito es usar el **procesador** de forma eficiente.

Para lograr ese objetivo, es necesario compartir el procesador. El sistema debe ser capaz de presentarle un medio a los usuarios que les haga creer que los procesos realmente se ejecutan a la vez.

El SO debe disponer de algún módulo que se encargue de decidir quién y en qué momento usará el procesador, ese componente se denomina genéricamente el **planificador**.

Figura I.4. La cola de listos controlada por el planificador de periodo corto. Fuente: Elaboración propia.

Para lograr sus objetivos, el SO debe mantener una o varias colas de planificación (figura I.4); también existen otras colas para competir por los diferentes recursos. Una de las colas más importantes en cualquier tipo de SO es la cola de listos, en ella hacen fila todos los procesos (representados por su PCB) que están listos para usar el procesador, es decir, los que pueden pasar del estado de listo al de ejecutando. En la figura I.4, los procesos P_3, P_1 y P_9 hacen la cola para usar el procesador; un planificador especial, denominado **planificador de periodo corto**, deberá decidir a cuál proceso se le asignará el procesador y esa decisión estará basada en una determinada política que establece algún algoritmo de planificación.

I.4.1 Los planificadores y el despachador

Dependiendo del SO, pueden existir varios planificadores (*schedulers*); en general, se distinguen tres: de periodo largo, de periodo corto y de periodo medio.

Planificador de periodo largo (*long term scheduler*)

En los SO de tratamiento por lotes (*batch*), es muy frecuente que se escojan más trabajos para incorporarlos al sistema que los que realmente pueden ejecutarse de inmediato. Esos trabajos se guardan en un equipo de almacenamiento masivo (típicamente un disco), donde se mantienen para su posterior ejecución. La tarea del planificador de periodo largo es escoger trabajos desde ese lugar y cargarlos a memoria para ponerlos como candidatos a ejecutar.

El planificador de periodo largo (conocido también como planificador de trabajos) se ejecuta con menor frecuencia que los demás planificadores y es el responsable de mantener una buena **mezcla de trabajos**, lo que significa que se combinen procesos que usan mucho el procesador con procesos que hacen muchas entradas salidas. Los procesos que usan mucho el procesador se denominan **acotados a CPU** y se caracterizan por tener largas ráfagas de CPU; los procesos que hacen muchas entradas salidas se conocen como **acotados a E/S** y se caracterizan por tener largas ráfagas de E/S.

Para lograr una buena mezcla de trabajo, se toma en cuenta que los procesos acotados a E/S usan poco la CPU, y cuando la necesitan, es por instantes pequeños de tiempo,

posiblemente para tomar algunas direcciones y después continuar en su tarea de E/S, que es atendida por los controladores de los equipos sobre los cuales se realiza la acción. Por eso se pueden favorecer o privilegiar los procesos que están acotados a E/S para asignarles el procesador que liberarán enseguida. En la figura I.5, se puede observar la interacción del planificador de periodo largo con el equipo de almacenamiento masivo (donde están almacenados los trabajos) y la cola de listos.

En el caso de que exista, este planificador será el encargado de la transición de nuevo a listo.

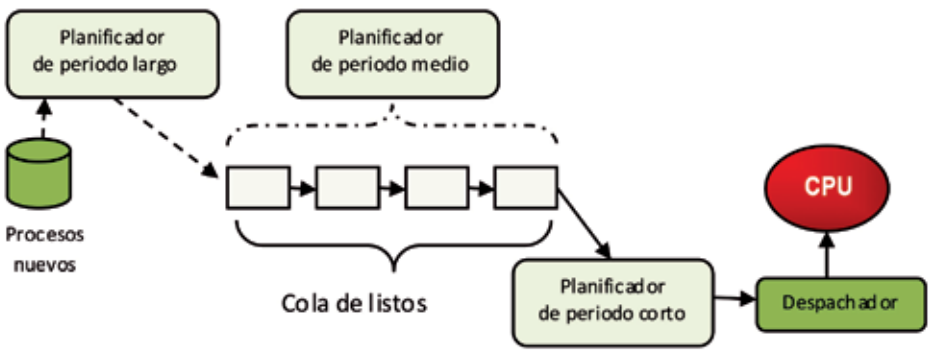


Figura I.5. Los planificadores y el despachador. Fuente: Elaboración propia.

El planificador de periodo corto (*short term scheduler*)

Su tarea es escoger de la cola de listos cuál será el próximo proceso a ejecutar para después comunicarle su decisión al otro componente del sistema de planificación –denominado **despachador**– (observe la figura I.5).

Este planificador (conocido, además, como **planificador del procesador**) se ejecuta con mucha frecuencia y por eso su código debe ser eficiente. La transición de listo a ejecutando es responsabilidad de este planificador.

El planificador de periodo medio (*medium time scheduler*)

Algunos SO adicionan este planificador, que tiene la función de disminuir el grado de multiprogramación (figura I.5). Para lograr ese objetivo, el SO retira algunos procesos de la competencia por el procesador, después de lo cual debe observar si hay mejorías que permitan introducirlo de nuevo (en ese caso los pone en la cola de listos). Es claro que si existen muchos procesos compitiendo por el procesador, la cola de listos se alarga produciendo un fenómeno conocido como **efecto de *convoy***; así, los procesos pueden verse afectados por el tiempo que pasan inactivos esperando a que se les asigne el procesador, de ahí que el trabajo de este planificador permita reducir ese problema.

El despachador (*dispatcher*)

El despachador es otro componente del SO y tiene la responsabilidad de garantizar la ejecución futura del proceso saliente (el que tenía la CPU) y de darle el control de la CPU

al proceso entrante (escogido por el planificador de periodo corto), es decir que debe efectuar lo que se conoce como **cambio de contexto**, para lo cual realiza las acciones siguientes:

1. Guarda en el PCB del proceso interrumpido (estaba en el estado ejecutando) los valores del procesador y toda la información necesaria para reiniciar el proceso en un futuro.
2. Copia en los registros del procesador, y desde el PCB del proceso escogido por el planificador, los valores necesarios para continuar la ejecución de ese proceso en el mismo lugar donde fue interrumpido en el pasado.
3. Cambia el procesador a modo usuario.
4. Salta a la localización de la próxima instrucción a ejecutar (un campo del PCB contiene el valor del contador de programa cuando fue interrumpido ese proceso).
5. Le da el control al proceso que entró para que continúe su ejecución.

En muchos SO, solo existe un planificador que se encarga de todas las tareas de planificación y despacho.

I.4.2 Objetivos de la planificación de la CPU

La planificación de la CPU es la tarea primordial en los SO multiprogramados, y los planificadores se encargan de realizarla de manera que el procesador se comparta entre los distintos procesos que se ejecutan en el sistema.

Existen diversos criterios que deben tomarse en cuenta para comparar los algoritmos de planificación de la CPU; entre otros se pueden citar:

1. Eficacia. Se trata de mantener la CPU utilizada el mayor tiempo posible.
2. Equidad. Garantizar que cada proceso obtenga un tiempo de CPU que se pueda considerar justo.
3. Tiempo de respuesta. Minimizar el tiempo en que comienzan a dar respuesta los procesos interactivos y el tiempo que deben esperar los usuarios de procesamientos por lote.
4. Rendimiento. Maximizar la cantidad de tareas procesadas por unidad de tiempo.

Es prácticamente imposible lograr todos esos objetivos en conjunto, de ahí que la mayoría de las veces se asume una solución de compromiso que toma en cuenta las prioridades de lo que se desea obtener.

I.5 OPERACIONES SOBRE PROCESOS

El SO —de forma directa— y los usuarios —a través del SO— pueden realizar diversas operaciones sobre los procesos; entre ellas se pueden citar las siguientes.

Creación de proceso

Un proceso —denominado "padre"— puede crear varios procesos —denominados "hijos"—, los cuales a su vez pueden tener otros hijos formando un árbol de procesos.

La figura I.6 muestra un árbol de procesos. El proceso P_{12} se ha bifurcado en dos procesos: P_{13} y P_{14}; a su vez, el proceso P_{13} crea tres procesos: P_{15}, P_{16} y P_{17}; en este caso, todos los procesos tienen un ancestro común que es P_{12} (se encuentra en la raíz del árbol de procesos). Cuando un proceso crea un hijo, existen dos posibilidades en relación con la ejecución:

* El padre continúa ejecutando concurrentemente con el hijo.
* El padre espera a que el hijo termine.

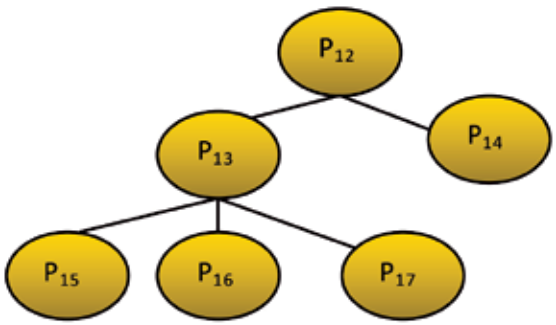


Figura I.6. Árbol de procesos. Fuente: Elaboración propia.

También existen dos posibilidades en relación con la memoria:

* El proceso hijo es un duplicado del padre porque tiene el mismo código (programa) y datos que el padre. La llamada al sistema fork() de los SO tipo Unix tiene ese comportamiento: por ejemplo, si en el árbol de procesos de la figura I.6 los hijos P_{13} y P_{14} del proceso P_{12} se hubieran obtenido ejecutando fork(), los tres procesos (P_{12}, P_{13} y P_{14}) serían idénticos y solo se diferenciarían en su PID, aunque serían procesos distintos que ocupan zonas diferentes de memoria. El programador deberá tomar decisiones para dirigir el flujo del proceso padre a una parte del código y el del hijo a otra.
* El proceso hijo carga un nuevo programa. Esto es lo que ocurre con la llamada al sistema fork() y una posterior llamada al sistema exec() en los SO de la familia Unix; esta última llamada al sistema sustituye el código heredado por el nuevo código que se especifique.

Terminación de procesos

Un proceso termina cuando ejecuta su última sentencia y le pide al SO que lo elimine, para lo cual usa una llamada al sistema. El SO le retira todos los recursos asignados. Un proceso también puede terminar la ejecución de otro (si es que se le permite).

Con el propósito de comprender mejor estos conceptos, se presenta el programa[19] I.1, elaborado para el SO Unix, en el cual se usan dos llamadas al sistema: fork() y getpid() (la figura I.7 muestra la sintaxis de estas dos llamadas al sistema).

La llamada al sistema fork() crea un proceso nuevo y devuelve los valores siguientes:

* En el código que ejecuta el hijo, el número cero (0).
* En el código que ejecuta el padre, el identificador del proceso hijo (PID). El PID es un número entero mayor que cero.
* En el caso de que haya error, devuelve -1.

La llamada al sistema getpid() devuelve el PID del proceso que la invoca.

```
Llamada al sistema fork()
#include  <sys/types.h>
#include <unistad.h>

pid_t pid fork (void)
```

```
Llamada al sistema getpid()
#include  <sys/types.h>
#include <unistad.h>

pid_t getpid(void)
```

Figura I.7. Declaración de las llamadas al sistema fork() y getpid(). Fuente: Elaboración propia.

Debe observarse que antes de la ejecución de la llamada al sistema fork() solo existe un proceso y que la sentencia: child = fork() se ejecuta de derecha a izquierda, es decir:

1. Primero se ejecuta fork(), que crea un nuevo proceso, nombrado proceso hijo. El hijo es una copia exacta del padre, es decir, tiene el mismo código que el padre y hereda todas las variables con sus valores actuales pero en otro espacio de memoria, de modo que son variables nuevas (que tienen los mismos valores en ese momento).
2. Después se evalúa la sentencia de asignación (=).
   a. La sentencia fork() en el hijo devuelve 0, ese valor se le asigna a la variable child del hijo.
   b. La sentencia fork() en el padre devuelve un valor mayor que 0 (es el PID del hijo), que se le asigna a la variable child del padre.

19 Todos los programas del libro han sido probados en ambientes reales.

```
// Programa I.1
# include <stdio.h>
# include <sys/types.h>
# include <unistd.h>
main()
  {
    pid_t child;              //pid_t tipo de entero para contener los PID
    int i = 0;                //Variable auxiliar, con fines didácticos, para distinguir los procesos
    child = fork ();          //Se crea un nuevo proceso
    /*A partir de este punto existen dos procesos que ocupan distintos lugares de memoria
      y tienen PID diferentes. El código y los nombres de las estructuras de ambos procesos son
     idénticos */
    if (child < 0)            //Si fork() devolvió un valor negativo, hubo un error
      {
        fprintf(stderr, "\nError al crear el proceso\n");
      }
    if (child == 0)           //Este código lo ejecuta el hijo, ya que fork() devolvió 0
      {
        i = 1;
        printf("\nSoy el nuevo proceso: %d\n", getpid());
      }
    else                      //La variable child en el padre recibe el PID del hijo ≠ 0
      {
        i = 2;
        printf("\nSoy %d, ejecuto concurrentemente con %d\n", getpid(), child);
      }
    printf("\nEsto lo ejecuta %d, el valor de i es %d\n", getpid(), i);
  }
```

Es muy importante que el lector entienda lo anterior, es decir, después de ejecutarse la sentencia fork() existen dos procesos que tienen el mismo código y manipulan variables con el mismo nombre, pero en espacios de direcciones de memoria distintos (las variables son diferentes aunque se llamen igual, tal como son diferentes dos personas con el mismo nombre). Es por eso que la última sentencia printf() del programa I.1 se imprime dos veces (si no hubo error), pero por procesos distintos (el padre y el hijo) y con valores distintos para la variable i.

La figura I.8 muestra la idea general con un ejemplo basado en el programa I.1; en la figura se muestran dos procesos (1849 y 1851) y sus PCB. El proceso 1849, que ocupa una zona de memoria entre las direcciones 20 y 40, ha ejecutado la sentencia fork() y el contador de programa ha recibido el valor 3 para indicar que la próxima instrucción a ejecutar está en esa dirección de memoria (es una dirección relativa, referida al inicio de la memoria del proceso 1849, es decir, la 20).

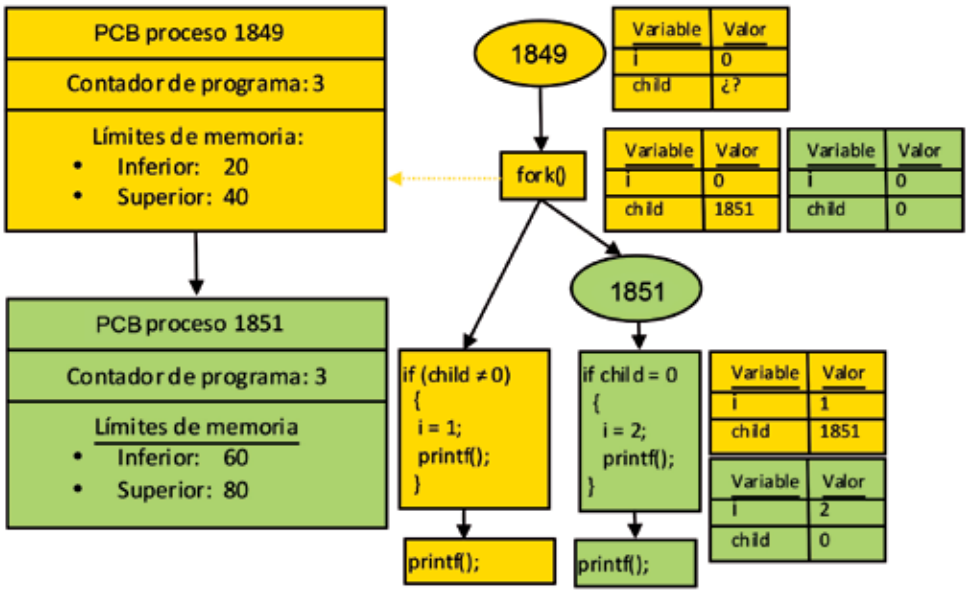


Figura I.8. Representación esquemática de la bifurcación con fork(). Fuente: Elaboración propia.

La ejecución de fork() hizo que se creara un proceso nuevo que es una copia exacta del proceso 1849 (el padre); al nuevo proceso (el hijo) se le asigna la memoria que está entre las direcciones 60 y 80, el PID 1851, y se crea un PCB que contiene los mismos valores del PCB del padre, excepto los datos de los límites de memoria, es decir que ocupa una zona distinta a la del proceso 1849.

En la figura I.8, el lector debe observar los colores que se han usado para distinguir los procesos, así como los valores de las variables en cada momento de la ejecución de 1849 y 1851. Obsérvese que la próxima instrucción a ejecutar de ambos procesos después del fork() tiene la misma dirección relativa (3), siendo las direcciones reales: 20+***3*** para 1849 y 60+***3*** para 1851.

Ejercicios

1. Ponga a punto el programa[20] I.1. Analice sus salidas y asegúrese de que lo entiende.
2. Use la ayuda del SO Unix (comando man) e incluya, en el programa I.1, las llamadas al sistema getppid() y exit(). La primera para obtener el PID del proceso padre y la segunda para terminar un proceso.

I.6 COMUNICACIÓN ENTRE PROCESOS (IPC)

Los procesos que se ejecutan en un SO pueden ser independientes o pueden comunicarse entre sí con el propósito de realizar tareas cooperadas. Los procesos que cooperan entre sí necesitan alguna forma de comunicación; existen, básicamente, dos formas.

20 Poner a punto un programa es un proceso que comprende las etapas de: edición, compilación, corrección de errores, ejecución y prueba.

* La primera estrategia para comunicar procesos se logra compartiendo una memoria común. Es un método rápido, pero trae asociadas otras complejidades. Obsérvese, en la figura I.9, que es crucial el momento en que el proceso P_1 lea el valor de la variable A; si lo hace antes de que P_0 ejecute la sentencia A = 10, el valor que leerá será 0 y si lo hace después, será 10. Esta problemática se analizará más adelante.

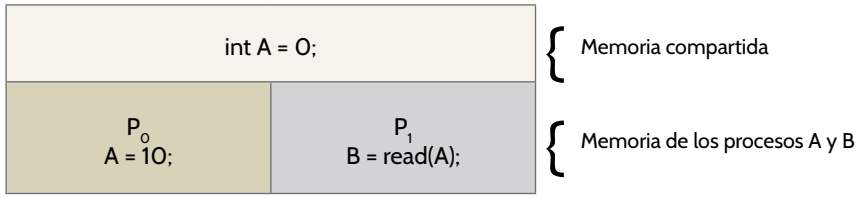


Figura I.9. Comunicación a través de memoria compartida.

* La segunda estrategia para comunicar procesos es a través de mensajes (figura I.10). Esta forma de comunicación es útil para intercambiar pequeños volúmenes de información y resulta fácil de implementar. Obsérvese que se necesita una cierta sincronización y que los mensajes se pueden perder (el envío o el acuse de recibo).

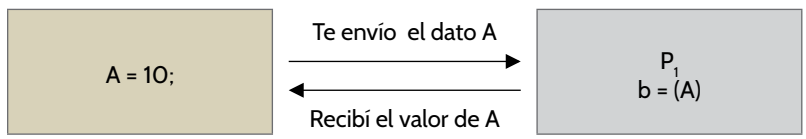


Figura I.10. Comunicación a través de paso de mensaje. Fuente elaboración propia.

Comunicación en sistemas de memoria compartida

En un sistema de memoria compartida, es necesario que los procesos establezcan una vía de comunicación para definir cuál es la parte de la memoria que se va a compartir.

Es muy bueno aclarar que el SO, normalmente, no permite que un proceso **A** acceda a la memoria de otro proceso **B** (y viceversa), lo cual es una forma de protegerlos. Por tal motivo, los procesos establecen ese acuerdo y de ahí en adelante el SO no participa en esa comunicación.

El problema del productor-consumidor

Este es uno de los problemas clásicos para hacer notar la complejidad de compartir algo, en especial la memoria. Se conoce con este nombre debido a que se basa en **dos procesos** que **comparten una zona de memoria**. Uno de ellos, denominado **productor**, produce información y la pone en la **memoria compartida** para que el otro, llamado **consumidor**, la extraiga de la memoria común y la procese (la consuma).

El productor puede estar produciendo mientras el consumidor consume, con el único cuidado de no estar usando exactamente las mismas direcciones de la memoria compartida, la cual se puede ver como un búfer con diferentes elementos.

Si se considera un búfer acotado (tiene un límite), se pueden hacer las siguientes apreciaciones:

* El productor tiene que esperar si el búfer está lleno. Cuando el consumidor consuma algo, liberará algún espacio.
* El consumidor tiene que esperar si el búfer está vacío. Cuando el productor produzca algo y lo ponga en el búfer, habrá algo que consumir.

De todo lo anterior, queda claro que debe existir una sincronización entre ambos procesos; por ahora se deja este problema abierto.

I.7 HILOS O PROCESOS LIGEROS

Los procesos que se han analizado hasta el momento se conocen también por el nombre de **procesos pesados.** La calificación de "pesado" viene dada por el hecho de que crear este tipo de proceso resulta una tarea compleja. Los procesos pesados, además, no comparten nada y la única forma de comunicarse entre sí es a través del **paso de mensajes** o por otras técnicas tales como las tuberías (se verán en el capítulo V), pero sin usar memoria común porque no la comparten.

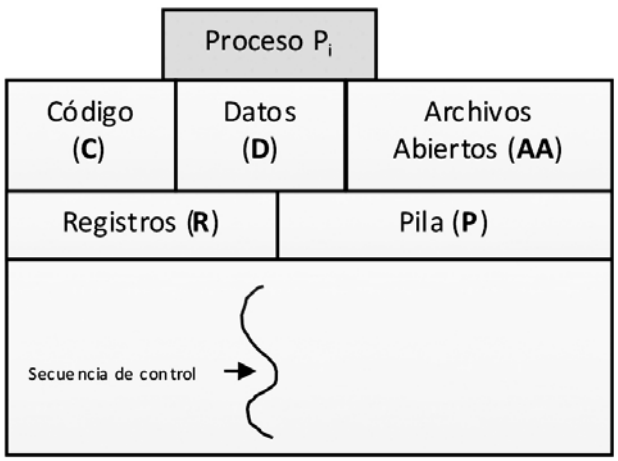


Figura I.11. Modelo de proceso. Fuente elaboración propia.

El modelo de proceso pesado —tal como se analizó en el apartado anterior— posee una **secuencia simple** de control de ejecución (figura I.11); es decir, sus instrucciones se ejecutan una detrás de otra de acuerdo con un orden establecido por el programador y los valores de las variables que controlan el flujo de ejecución.

Un hilo (thread) también se puede definir como **un proceso en ejecución** y **una unidad de planificación y despacho**, pero los hilos son hijos de algún proceso y comparten un conjunto de atributos comunes que facilitan, entre otras cosas, la comunicación a través de una memoria común.

Entonces se infiere que un proceso puede tener varios hilos (o **procesos ligeros**). Cada hilo tiene su propia secuencia de control de ejecución, lo que permite que todos los hilos del mismo proceso se puedan ejecutar en forma concurrente. Obsérvese entonces que un proceso desde el cual se bifurcan varios procesos ligeros (hilos) posee varias secuencias de control (una por cada hilo) que se ejecutan de forma independiente una de otra.

Cada hilo necesita de un **PCB** que, en este caso y para no confundir, se podría denominar Bloque de Control de Hilo, pero que en realidad tiene la misma estructura que el PCB visto antes, y de hecho el planificador y el despachador no distinguen diferencias entre ellos. Se puede decir que el hilo es la menor unidad de planificación y despacho porque el planificador planifica hilos.

En el modelo de hilo, todos los hilos, que son hijos de un mismo proceso, comparten el código, los datos globales y los archivos abiertos; pero cada hilo tiene su propia copia de los registros del procesador y una pila, además de una secuencia de control, también propia. Esta concepción permite que un proceso pesado que se bifurca en varios procesos ligeros pueda ejecutar distintas partes del código compartido.

En la figura I.12, los hilos o procesos ligeros H_1 y H_2 son hijos del proceso pesado P_i. Los hilos H_1 y H_2 tienen acceso al mismo código (**C**), a los datos globales (**D**) y a los archivos abiertos (**AA**) que heredan del padre, pero:

* El hilo H_1 tiene su propia copia de los registros del procesador ($\mathbf{R}_1$), además de una pila particular ($\mathbf{P}_1$), lo cual le permite tener una secuencia de ejecución, $\mathbf{S}_1$, propia que ejecuta un código distinto al hilo H_2.
* El hilo H_2 tiene su propia copia de los registros del procesador ($\mathbf{R}_2$), además de una pila propia ($\mathbf{P}_2$) y por eso tiene una secuencia de ejecución, $\mathbf{S}_2$, propia, que ejecuta un código distinto al hilo H_1.

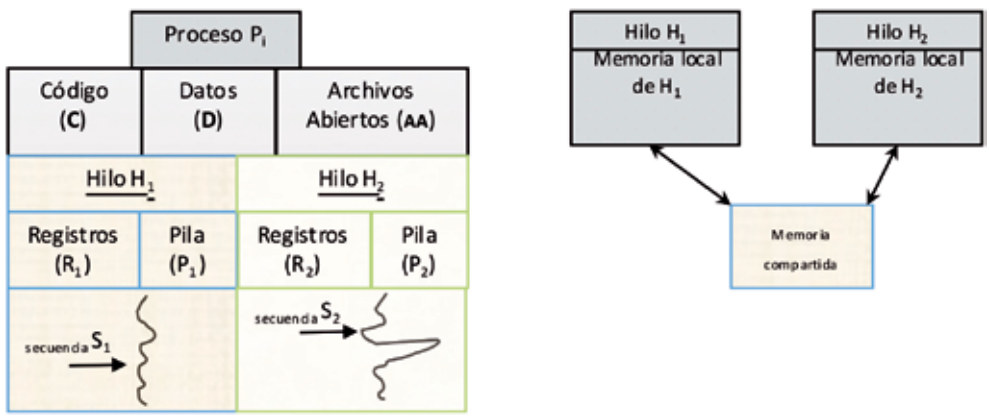


Figura I.12. Modelo de hilo. Fuente elaboración propia.

Debe observarse la parte derecha de la figura, en la cual se aprecia que cada hilo tiene su propia zona de memoria en la que residen los datos que se han declarado de forma local, pero ambos comparten una memoria común con el padre.

I.7.1 Beneficio de los hilos

Entre otras aplicaciones, los hilos se usan en la implementación de servidores de red y servidores web. También favorecen la ejecución paralela de aplicaciones en sistemas de multiprocesadores de memoria compartida. Los beneficios de la programación multihilo se pueden ver desde cuatro aristas:

* Mejor respuesta.

Una aplicación multihilo permite que el proceso continúe ejecutando aun cuando alguna de sus partes está bloqueada o está resolviendo una operación muy lenta. Por ejemplo, un navegador multihilo puede permitir que el usuario interactúe con él (a través de un hilo) mientras está cargando una imagen (usando otro hilo); también es posible salir adecuadamente de una aplicación, que se ha bloqueado indefinidamente, si posee un hilo independiente que permite hacer esa acción.

* Compartir recursos.

Como se aprecia en la figura I.12, los hilos comparten la memoria y los recursos del proceso al cual pertenecen. Los hilos H_1 y H_2 comparten la sección de datos de P_i y por eso la aplicación puede tener varios hilos en actividad dentro del mismo espacio de direcciones.

* Economía.

En el ejemplo de la figura I.11, se aprecia que los procesos pesados no comparten nada entre sí; por su parte, el ejemplo de la figura I.12 muestra que los hilos comparten la sección de código, datos y archivos abiertos. A partir de la observación anterior, se puede notar que es más rápida la asignación de recursos a un hilo nuevo que la asignación de recursos a un proceso pesado nuevo. Es decir, es más económica la creación de un hilo debido a que comparte los recursos con el proceso al cual pertenece, y también es más ágil el cambio de contexto entre hilos de un mismo proceso que el cambio de contexto entre dos procesos pesados. Por todo esto, los hilos se denominan **procesos ligeros**.

Así mismo, debe señalarse que para que dos procesos se puedan comunicar, el núcleo del SO debe intervenir (por un problema de protección), cosa que no ocurre cuando dos hilos se comunican a través de la memoria que comparten.

Otro detalle digno de destacar es que la terminación de un hilo es más ágil que la terminación de un proceso.

* Utilización de las arquitecturas de multiprocesadores.

Los beneficios de la programación con hilos pueden incrementarse considerablemente en un sistema con más de un procesador, dado que los hilos pueden ejecutarse en paralelo de forma real y hoy en día ya son populares las máquinas con más de un procesador o con varios núcleos en el mismo procesador (multinúcleo).

Muchos SO modernos usan el concepto de hilo; por ejemplo, Solaris crea un conjunto de hilos en el *kernel* para controlar las interrupciones, Linux usa un hilo del *kernel* para controlar la cantidad de memoria, etc.

El soporte para hilo se da a dos niveles:

1. A nivel de usuario (hilos de usuario). En ese caso, no reciben ningún apoyo del SO, sino que existe una biblioteca de hilos que contiene el código necesario para crearlos, destruirlos, pasar mensajes y datos entre los hilos, planificarlos, salvar y restaurar el contexto.

2. A nivel del núcleo o *kernel* (hilos de *kernel*). Este tipo de hilo los maneja directamente el SO, la planificación se hace a nivel de hilo y el *kernel* es el responsable de crearlos, planificarlos y manejarlos en el espacio del *kernel*. Generalmente, la creación y manipulación de este tipo de hilo es más lenta que la manipulación de los hilos de usuario.

Casi todos los SO modernos incluyen soporte para hilos a nivel de *kernel*, por ejemplo: Windows, Linux, Mac OS x y Solaris, entre otros.

Un ejemplo que ayuda a comprender la importancia de los hilos es el siguiente:

Un profesor de SO está haciendo un uso intensivo de un aula de computadoras y de la red, y todos los estudiantes están accediendo a un servidor web dado, lo que genera una significativa cantidad de solicitudes que el servidor debe satisfacer: ¿qué sucedería si una aplicación con un solo hilo de control fuera la responsable de satisfacer todas esas solicitudes? La respuesta es obvia, la cola de espera sería muy grande y la prestación de servicios sería muy mala.

Si ese mismo acceso fuera a nivel global, por ejemplo, el acceso a Google que diariamente hacen muchas personas en el mundo, el problema se agravaría notablemente (claro que Google no toma una decisión tan incorrecta).

El modelo de hilo puede resolver parte de ese problema al permitir que para cada petición se cree un hilo que "escucha" y atiende las solicitudes de forma independiente.

En esencia, los hilos tienen las siguientes características:

1. Todo hilo existe dentro de un proceso y usa los recursos de ese proceso.

2. Cada hilo tiene su **propio flujo de control**, que es independiente del flujo de control del padre y de sus hermanos.

3. El hilo solo duplica los recursos esenciales que le permiten ser planificado de forma independiente.

4. El hilo debe compartir los recursos del proceso padre con otros hilos del mismo proceso.

5. Se dice que un hilo es un proceso ligero debido a que la mayoría de la sobrecarga que significa la creación de un proceso ya fue hecha cuando se creó su proceso padre.

I.7.2 Bibliotecas de hilos

Existen algunas bibliotecas[21] que permiten manejar hilos y brindan a los programadores una interfaz que da acceso a diferentes funciones para manipularlos. Esas bibliotecas pueden residir en el espacio de usuario o en el espacio que ocupa el núcleo del SO, de modo que:

* Cuando se invoca una función de biblioteca que reside en el **espacio de usuario**, se está accediendo a una **función local** dentro de la **memoria que manipula la aplicación**.
* Cuando se invoca a una función de biblioteca que reside en el **núcleo del SO**, se está realizando una **llamada al sistema**, lo que implica el **cambio de contexto del SO** de modo usuario a modo protegido.

Existen tres bibliotecas importantes que se apoyan en estas ideas: Pthreads, Win32 y Java. La biblioteca de hilos de Java solo se mencionará sin ofrecer más detalles porque el objetivo del libro no se enfoca en la programación en general.

Hilos con Pthread

Pthread es una **API** (Application Program Interface) para la creación y sincronización de hilos. En realidad, es una especificación para hilos (según el estándar POSIX[22] IEEE 1003.1) y no una implementación, lo que da libertad a los diseñadores de SO para implementarla de acuerdo con sus necesidades.

Las implementaciones que se adhieren a este estándar la refieren como POSIX Threads o Pthreads. Muchos SO implantan Pthreads, por ejemplo: Solaris, Linux, Mac OS X, etc.

Existen diversas funciones dentro de la biblioteca Pthreads, a continuación se mencionan algunas de sus acciones:

* Crear, separar y unir hilos.
* Fijar u obtener los atributos de los hilos.
* Manejar la exclusión mutua o *mutex* (*mutual exclusion*), garantizando que el acceso a algunas partes del código sea exclusivo (solo un hilo puede estar en un instante de tiempo dado ejecutando el código mencionado).
* Garantizar la sincronización entre hilos, etc.

Los nombres de las funciones de la biblioteca Pthread comienzan por el prefijo pthread_. Para comprender los ejemplos que se presentarán más adelante, se explican algunas de esas funciones:

21 Muchos autores usan la palabra "librería" para referirse a las bibliotecas de programas a causa de una mala traducción de la palabra *library* y el error se ha generalizado en exceso.

22 POSIX es un acrónimo de Portable Operating System Interface, una norma de la IEEE que define una interfaz estándar para desarrollar sistemas operativos.

1. Crear un nuevo hilo:

```
int pthread_create(pthread_t *thread, const pthread_attr_t *attr, void
*(*start_routine)(void*), void *arg)
```

La función pthread_create crea un nuevo hilo con identificador thread (es el equivalente al PID pero para hilos) y con atributos attr (si el puntero attr es NULL, se usan los atributos por defecto).

El hilo creado ejecutará la función start_routine, la cual toma el parámetro arg como argumento. Cuando dicha función retorne, el hilo finaliza.

2. Unir hilos:

```
int pthread_join(pthread_t thread, void **value_ptr)
```

La función pthread_join suspende la ejecución del hilo que la invoca hasta que el hilo thread termine (no tiene efecto si thread ya terminó). La localización apuntada por value_ptr permite analizar el código de terminación.

Las funciones pthread_create y pthread_join devuelven: cero (0) cuando tienen éxito y un entero diferente de cero en otro caso (ese entero indica algún tipo de error).

3. Identificar un hilo:

```
pthread_t pthread_self(void)
```

La función pthread_self() devuelve el identificador del hilo que la invoca.

El programa del ejemplo I.2 comienza con un solo hilo (una sola secuencia simple de control de ejecución o torrente de instrucciones a ejecutar), después de lo cual:

1. Usa la sentencia pthread_create(&tid, NULL, threadFun, NULL) para crear un segundo hilo (un segundo torrente de instrucciones que se ejecutará concurrentemente con el torrente del primer hilo). El identificador del hilo nuevo queda almacenado en la dirección de la variable tid, tendrá los atributos por defecto (debido a que el segundo parámetro es NULL) y ejecutará la función threadFun(), a la cual no se le pasan argumentos (el último parámetro es NULL).
2. Una vez creado el segundo hilo, el primero imprime su identificación por segunda vez y se queda en espera de que el nuevo hilo termine, para lo cual ejecuta la función pthread_join(tid, NULL), especificando que esperará por el hilo tid, el cual finaliza cuando termina de ejecutar la función threadFun().

```
Programa I.2
#include <pthread.h>                    // Incluir la biblioteca Pthread
#include <stdio.h>                      // Incluir la biblioteca de entrada/salida estándar
/*-----------------------------------------------------------------*
*La función threadFun() será invocada por el hilo creado*
*-----------------------------------------------------------------***/
void  *threadFun(void *arg)
  {
    printf("Saludos desde el hilo %u \n", (unsigned)pthread_self());
    return 0;
  }

//Programa principal
int main(void)
  {
    pthread_t tid;                      // Variable para identificar al hilo creado

    printf("En este momento solo existe el hilo %u\n", (unsigned)pthread_self());
    pthread_create(&tid, NULL, threadFun, NULL);
    printf("Ahora el hilo %u, ejecuta concurrentemente con el hilo %u\n",
           (unsigned)pthread_self(), (unsigned)tid);
    pthread_join(tid, NULL);        // se unen ambos hilos. De nuevo existe un solo hilo
    return 0;
  }
```

Si el programa anterior se guarda en un archivo denominado progI_2.c, el comando para compilarlo desde una terminal (en ambientes Unix, Linux) sería el siguiente:

```
gcc -o hilosE progI_2.c -lpthread
```

La opción -o especifica el nombre del ejecutable que se obtendrá (hilosE en este caso) y la última opción es para que el código del programa obtenido, hilosE, se enlace con la biblioteca lpthread.

Ejercicios:

1. Edite el programa anterior, compílelo y ejecútelo en una terminal. Analice la salida del programa.
2. Introduzca un ciclo en la función threadfun() y otro en el hilo principal y ejecute el programa. Analice la salida del programa.

Hilos con Win32

La API Win32 está disponible en los SO Windows. La función para crear los hilos, usando C++, es CreateThread() que se aprecia a continuación.

```
HANDLE WINAPI CreateThread
  (
    _In_opt_    LPSECURITY_ATTRIBUTES     lpThreadAttributes,
    _In_        SIZE_T                    dwStackSize,
    _In_        LPTHREAD_START_ROUTINE    lpStartAddress,
    _In_opt_    LPVOID                    lpParameter,
    _In_        DWORD                     dwCreationFlags,
    _Out_opt_   LPDWORD                   lpThreadId
  );
```

Los parámetros tienen las siguientes especificaciones:

* lpThreadAttributes. Puntero a una estructura SECURITY_ATTRIBUTES que determina si los procesos hijos pueden heredar el manipulador (*handle*) retornado. Si es NULL, no se puede heredar.
* dwStackSize. Longitud inicial de la pila en bytes; si es cero, el hilo usa el valor por defecto.
* lpStartAddress. Puntero a la función que ejecutará el hilo.
* lpParameter. Puntero a una variable (opcional) que se le pasa al hijo.
* dwCreationFlags. Bandera para controlar la creación del hilo de acuerdo con los valores siguientes:
  - 0: el hilo se ejecuta inmediatamente después de su creación.
  - create_suspended (0x00000004): el hilo queda en el **estado suspendido** hasta que se invoque la función ResumeThread().
  - stack_size_param_is_a_reservation (0x00010000): si esta bandera no se especifica, el tamaño de la pila es el especificado en el parámetro dwStackSize.
* lpThreadId. Puntero que recibe el identificador del hilo.

El programa I.3 muestra un ejemplo de creación de hilos usando la API WIN32. Se han situado diversos comentarios en el programa, sin embargo, resulta adecuado hacer algunas observaciones:

* La función principal, main(), construye MAX hilos usando la línea de código: hThreadArray[i] = CreateThread(NULL, 0, threadFun, &i, 0, dwThreadId [i]); de esta forma, los manipuladores de los hilos quedan almacenados en el arreglo hThreadArray.
* La función threadFun() usa la función GetCurrentThreadId() para obtener el identificador del hilo que la está ejecutando.
* El hilo principal invoca la función WaitForMultipleObjects() para esperar por los hilos creados. Después de que el hilo principal termina la espera, usa la función CloseHandle() para cerrar los manipuladores que abrió.

```
//Programa I.3
#include <stdio.h>
#include <windows.h>

#define MAX 3

DWORD WINAPI threadFun(LPVOID lpData)
  {
    unsigned int &counter = *((unsigned int *) lpData);
    printf("Soy el hilo %d, la variable recibida tiene el valor %d ",
          GetCurrentThreadId(), counter);
    return 0;
  }

int main(int argc, char *argv[])
  {
    HANDLE  hThreadArray[MAX];

    //Se crean MAX hilos
    for( int i = 0; i < MAX; i++ )
      {
        //Cada hilo ejecutará la función threadFun() que recibirá el valor de i
        //Los manipuladores de los hilos se guardan en el arreglo hThreadArray
        hThreadArray[i] = CreateThread(NULL, 0, threadFun, &i, 0, NULL);
        if (hThreadArray[i] == NULL)          //Si no se puede crear el hilo en la iteración i
          {
            printf("Error al crear el hilo");
            return 2;                         //Salir con código de error 2
          }
      }                                       //Fin de la creación de los hilos
    //El proceso padre espera por los hilos creados
    WaitForMultipleObjects(MAX, hThreadArray, TRUE, INFINITE);
    for( int i=0; i< MAX; i++ )
      {
        closeHandle(hThreadArray[i]);         //Cerrar los manipuladores abiertos
      }
    return 0;                                 //Salir sin error
  }
```

Un estudio más profundo de estas facilidades no está entre los objetivos del libro, por eso el lector interesado debe profundizar en la bibliografía que trata ampliamente esos aspectos.

A continuación, se aprecian las definiciones sintácticas de las funciones de la biblioteca WIN32 que se usaron en el programa I.3, GetCurrentThreadId y WaitForMultipleObjects, así como los detalles de la última función.

```
DWORD WINAPI GetCurrentThreadId(void);
DWORD WINAPI WaitForMultipleObjects
  (
    _In_   DWORD          nCount,
    _In_   const HANDLE   *lpHandles,
    _In_   BOOL           bWaitAll,
    _In_   DWORD          dwMilliseconds
  );
```

Especificidades de la función WaitForMultipleObjects():

* nCount. Cantidad de objetos manipulados; en el programa I.3 es MAX.
* lpHandles. Un arreglo de los objetos manipulados; en el programa I.3 es hThreadArray.
* bWaitAll. Si este parámetro es **TRUE**, como en el programa I.3, la función retorna cuando el estado de todos los objetos del arreglo lpHandles estén señalizados[23]. Si es **FALSE**, la función retorna cuando el estado de algunos de los objetos esté señalizado, y en ese caso, el valor retornado indica cuál fue el objeto que provocó el retorno de la función.
* dwMilliseconds. Indica el intervalo de tiempo, en milisegundos, que la función esperará.
  - Si el valor es diferente de cero, la función esperará hasta que el objeto esté en el estado de señalizado o hasta que termine el intervalo.
  - Si es **INFINITE**, como el programa I.3, la función solo retornará cuando el objeto sea señalizado.

I.8 ALGORITMOS DE PLANIFICACIÓN

Los algoritmos de planificación no hacen distinción entre procesos ligeros y pesados, ya que para ellos solo existen los procesos en su concepción general, de ahí que en la cola de listos se puedan encontrar procesos de ambos tipos, y el planificador de periodo corto escogerá alguno sin hacer distinciones entre uno u otro.

Las políticas de planificación de los SO pueden ser:

* Sin desalojo. En este tipo de política, a un proceso que está en el estado **ejecutando** no se le retira la CPU mientras esté usándola. El planificador de periodo corto solo entra en acción cuando el proceso que tiene el procesador necesita hacer una entrada/salida o cuando termina. Los SO Windows 3.x usaban esta forma de planificar.
* Con desalojo. Este segundo tipo de política permite que el SO le retire el procesador al proceso que lo tenga. De tal forma, a los dos momentos anteriores para planificar un nuevo proceso (terminó, tuvo que hacer una entrada/salida), se le agrega una tercera que es cuando se le quita el procesador al proceso que lo tiene, a pesar de que

23 "Señalizado" significa que se ha hecho una operación sobre el objeto, en este caso sobre el hilo.

necesitaba más tiempo. Los motivos para quitarle el procesador pueden ser varios, por ejemplo: se le había asignado el procesador por un tiempo y ese tiempo terminó, llegó un proceso de mayor prioridad, etc.

En cualquier caso, cuando un proceso **A** abandona la CPU, voluntaria o involuntariamente, el planificador de periodo corto debe escoger otro proceso **B** para asignársela. A continuación, se explican algunos de los algoritmos que se usan con ese propósito.

Algoritmo FCFS (First Come First Served[24])

Este algoritmo le asigna la CPU al primer proceso que haga la solicitud. Es el más simple de todos, pero su rendimiento puede ser bajo. Para implementarlo, se puede usar una cola **FIFO** (First In First Out[25]), es decir, una cola donde:

* Los procesos se insertan al final.
* Cada vez que se necesita escoger un proceso para ejecutar, se toma el que está en la cabeza de la cola y se elimina el proceso escogido de la cola.

Este algoritmo es, típicamente, sin desalojo dado que no existen criterios para retirarle la CPU al proceso que esté ejecutando. En este aspecto, es importante recordar que los procesos no están siempre haciendo trabajos en la CPU y que su vida transcurre en **ráfagas** de uso de la CPU y de E/S.

Para analizar el rendimiento de este y los restantes algoritmos, se pueden usar los diagramas de Gantt.

La figura I.13 muestra una tabla con los tiempos que necesita consumir cada proceso en su próxima etapa de ejecución (en milisegundos). A la derecha de la figura, se observan los diagramas de Gantt con los tiempos que tendrá que esperar cada proceso para iniciar su próxima etapa de ejecución, de acuerdo con el orden de llegada:

Procesos	Próxima etapa de CPU (en ms)	Orden de llegada a la cola	
		P_1, P_0, P_4, P_3	P_4, P_3, P_0, P_1
		Tiempos de espera (ms)	
P_0	4	15	5
P_1	15	0	9
P_3	2	21	2
P_4	3	19	0
Promedio		13,75	4

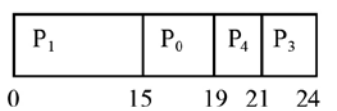


Diagrama de Gantt para orden de llegada P_1, P_0, P_4, P_3

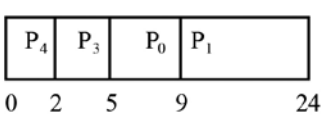


Diagrama de Gantt para orden de llegada P_4, P_3, P_0, P_1

Figura I.13. Análisis del algoritmo FCFS. Fuente elaboración propia.

24 Se traduce como "el primero que llega es el primero en ser servido".

25 Se traduce como "el primero que llega es el primero en salir".

La tercera columna de la tabla de la izquierda refleja los datos obtenidos; se puede apreciar cómo fluctúa, marcadamente, el promedio del tiempo de espera de una situación a otra.

Otro problema del algoritmo se produce cuando un proceso que usa mucho la CPU, por ejemplo, un proceso de cálculo científico, toma el procesador por un periodo largo de tiempo y varios procesos que realizan muchas E/S terminan su entrada o salida del momento, pero necesitan hacer pequeñas acciones con el procesador para continuar. En ese instante, cada uno de los procesos acotados a E/S pasa al estado de listo y se incorporan a la cola de igual nombre, la cual aumentará de tamaño considerablemente.

Algoritmo por prioridad (*priority*)

En este caso, lo que se hace es asociar una prioridad a cada proceso y se escoge el más prioritario. La forma que usa el algoritmo para escoger el próximo proceso a ejecutar hace que los procesos que tienen prioridades bajas esperen mucho tiempo; se dice que "se mueren de hambre esperando por el procesador" y el fenómeno se conoce como **inanición** (*starvation*). Este tipo de algoritmo se puede implementar con desalojo o sin desalojo.

Una solución al problema de inanición es que el SO les suba la prioridad a los procesos que llevan mucho tiempo en el sistema, de forma que en un futuro más cercano les toque el procesador. Se dice que los procesos que se ganan ese derecho "han envejecido" y por eso se han ganado el derecho de ejecutar. La subida de la prioridad puede establecerse cada cierto tiempo.

Las prioridades se pueden definir interna o externamente:

* La prioridad interna se puede establecer a partir de alguna medida que se pueda calcular, por ejemplo: la cantidad de memoria que necesitan los procesos, la cantidad de archivos que han abierto, etc.

* La prioridad externa no se asocia con el SO y tiene que ver con alguna consideración ajena a él, por ejemplo: la necesidad de tener una respuesta rápida de un proceso específico, de qué departamento proviene el trabajo, etc.

Los algoritmos basados en prioridades se pueden implementar con desalojo y sin desalojo. Para el caso de que se use el desalojo, la prioridad de un proceso que arribe (llegue) a la cola de listos se compara con la prioridad del que está usando la CPU; si la prioridad del proceso que llega es mayor, entonces se desaloja al que estaba en la CPU para darle ese recurso al nuevo proceso, y el proceso desalojado se envía a la cola de listos.

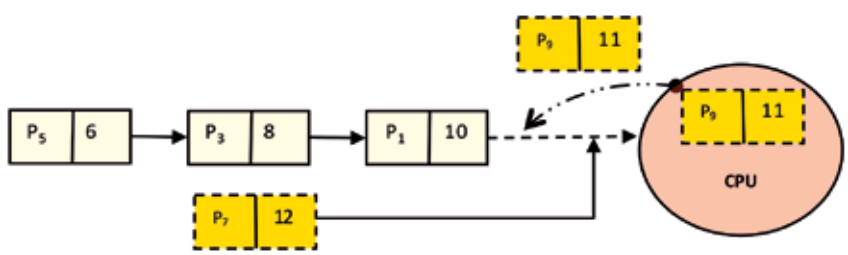


Figura I.14. Algoritmo de prioridades con cola ordenada. Desalojo del proceso P_9 y entrada del proceso P_7. Fuente elaboración propia.

La cola de listos, en este algoritmo, se puede mantener ordenada por prioridades de mayor a menor (el proceso más prioritario en la cabeza de la cola) o desordenada. Si la cola está ordenada (figura I.14), el planificador de periodo corto siempre escogerá el proceso que está en la cabeza, pero tendrá que ordenar la cola cada vez que llegue un proceso a ella; si la cola está desordenada, no tendrá ese trabajo, pero para seleccionar el próximo proceso tendrá que recorrer toda la cola.

En la figura I.14, la cola de listos está ordenada por prioridades. En el momento que se ejemplifica, está llegando a la cola el proceso P_7 con prioridad 12 y se está ejecutando el proceso P_9 con prioridad 11. EL SO compara las prioridades y se percata de que ha llegado un proceso nuevo con prioridad mayor al que está en la CPU, por lo cual desaloja a P_9, lo pone al frente de la cola y le asigna la CPU al proceso P_7. Si la prioridad del proceso nuevo no es mayor que la del proceso que está ejecutando, se insertará en el lugar que le corresponde en la cola (tendrá que recorrerse la cola hasta encontrar ese lugar).

Obsérvese que si la cola no está ordenada, el proceso nuevo se puede insertar en cualquier lugar, pero es más fácil situarlo al final.

Algoritmo SJF (Shortest Job First[26])

El tercer algoritmo que se analiza toma en cuenta la longitud de la próxima ráfaga de CPU de los procesos que compiten por el procesador, es decir que prioriza al que tenga la próxima necesidad de CPU más corta (¿cuál proceso necesitará menos tiempo?). Este algoritmo es un caso particular derivado del anterior.

Si se hace un análisis similar al que se hizo en el algoritmo FCFS, SJF arroja resultados muy alentadores y puede decirse que posiblemente sea óptimo. El problema es que no hay forma de conocer la longitud de la próxima ráfaga de CPU que necesitarán los procesos, por eso el algoritmo no puede implementarse y solo se usa para tomarlo como punto de comparación o también se hacen aproximaciones a él.

El planificador de periodo largo de los sistemas de tratamiento por lotes (*batch*) podría usar SJF, para lo cual es posible aprovechar el hecho de que en este tipo de sistema los usuarios someten los lotes acompañados por una descripción general y una especificación de cada proceso; en esta última, se puede incluir una estimación del tiempo.

Existen algunas vías para estimar el tiempo del próximo intervalo de uso de la CPU de cada proceso que compite por el procesador; por ejemplo, ponderar los tiempos de ejecución que tuvieron los procesos en el pasado.

Algoritmo por torneo (Round Robin)

El algoritmo de torneo, más conocido por Round Robin, fue diseñado especialmente para SO de tiempo compartido, es decir, sistemas que comparten el tiempo de la CPU en periodos iguales que se denominan **quantum de tiempos**; ese tiempo generalmente está entre 10 y 100 milisegundos. Los procesos que arriban a la cola de listos entran al final de esa cola sin importar de dónde vengan.

26 Se traduce como "el trabajo más corto primero".

La idea del algoritmo es darle un tiempo igual a todos los procesos; una vez transcurrido ese tiempo (se dice que ha expirado el tiempo), se le retira el procesador sin importar si el proceso ha finalizado su ráfaga de CPU o no; el proceso desalojado se envía al final de la cola de listos.

De esta manera, se forma una cola circular y el planificador de periodo corto siempre toma al primer proceso de la cola para darle el procesador. Si un proceso abandona la CPU antes de haber finalizado su tiempo (por ejemplo, porque necesita realizar una E/S o ha terminado), se escoge de nuevo el primer proceso de la cola y se le asigna el mismo quantum de tiempo, es decir que nadie se aprovecha del tiempo no utilizado por el proceso saliente, lo único que ocurre es que el proceso al que le toca el turno comienza más temprano.

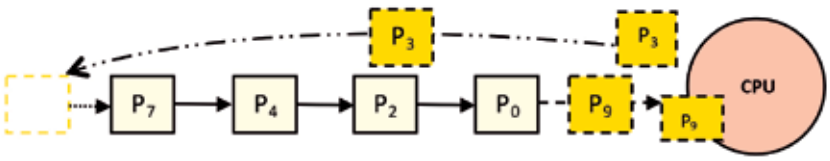


Figura I.15. Round Robin. Desalojo del proceso P_3 y entrada del proceso P_9. Fuente elaboración propia.

La figura I.15 muestra la idea general del algoritmo Round Robin; en ese instante, al proceso P_3 se le agotó el quantum de tiempo, por eso se desaloja y se envía al final de la cola, después de lo cual se toma el primer proceso de la cola (P_9) y se le asigna la CPU por el tiempo establecido (siempre es el mismo).

Para controlar el tiempo, el sistema dispone de un **temporizador** que se inicia, con el valor del quantum de tiempo, cada vez que un proceso entra a la CPU. El temporizador disminuye su valor en uno cada milisegundo, de forma que expira cuando llega a cero y en ese momento entra en escena el despachador. Debe observarse que en este caso la labor del planificador de periodo corto es prácticamente nula.

Algoritmos de colas múltiples

El algoritmo de colas múltiples divide la cola de listos en varias colas (en la figura I.16 son tres: A, B y C).

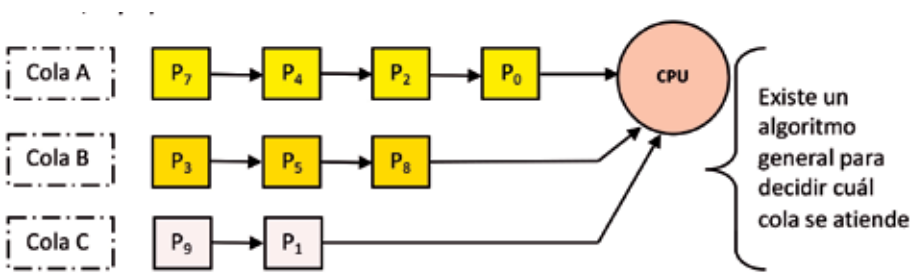


Figura I.16. Colas múltiples. Fuente elaboración propia.

Los procesos se asignan permanentemente a alguna de esas colas, para lo cual se toman en cuenta sus propiedades, que pueden ser la cantidad de memoria que necesita, la prioridad, el tipo de proceso, etc.

Cada cola tiene su propio algoritmo de planificación y existe un algoritmo general para todas las colas. Este último establece cómo se atienden las colas y puede ser que se prioricen por orden, de forma que las colas inferiores solo se atiendan cuando las superiores (de mayor prioridad) estén vacías.

Algoritmos de colas multinivel con retroalimentación

En este caso, a diferencia de las colas múltiples, los procesos pueden moverse de una cola a otra. Los procesos que arriban al sistema para competir por la CPU entran a una cola superior (el uso de la CPU determina por cuál cola entran), y en ella reciben un tiempo de CPU que es el menor de todos los que recibirán en las colas inferiores. Si un proceso **p** no termina en ese tiempo, se pasa a una cola inferior, en la cual recibirá un tiempo mayor de CPU, de modo que, según vaya migrando de arriba hacia abajo, se irá incrementando el tiempo de CPU hasta la última cola que se atiende FCFS, en donde se le da todo el tiempo que necesite en cada etapa (no se desaloja).

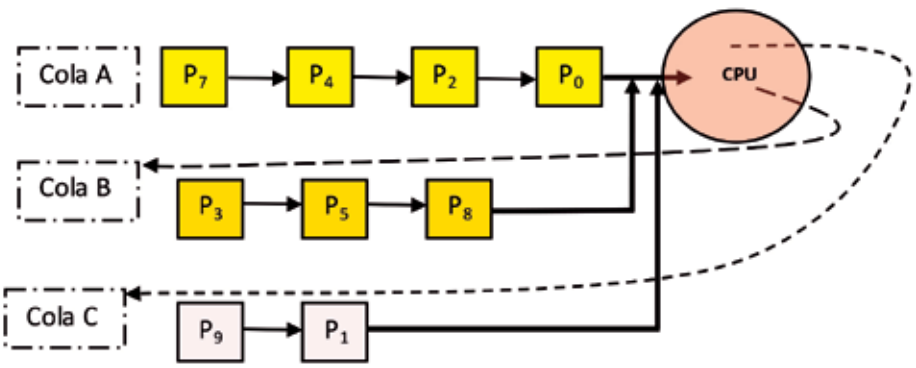


Figura I.17. Colas multinivel con retroalimentación. Fuente elaboración propia.

El algoritmo de planificación se hace de manera tal que solo atienda una cola inferior si las colas superiores están vacías. Es decir que en la figura I.17 primero se atiende la cola A, y cada vez que un proceso de esa cola (P_0, P_2, P_4 y P_7) reciba su tiempo, se pasa a la cola B, la cual solo se atenderá cuando la cola A esté vacía (si hay nuevos arribos a la cola A, habrá que atenderlos antes que los de la cola B). En la cola B, los procesos reciben un tiempo mayor, después de lo cual se desalojan a la cola C, la cual se atiende FCFS, es decir que se le da todo el tiempo que necesite en cada etapa sin desalojarlo.

Si se está atendiendo una cola inferior (B o C en el ejemplo) y llegan procesos a las colas superiores, se desaloja el proceso de la cola inferior y se pasa a atender el de la cola superior. Para tratar de resolver el problema de inanición, se permite que el planificador haga ascender trabajos que llevan demasiado tiempo en el sistema (han envejecido).

I.9 SINCRONIZACIÓN DE PROCESOS

El concepto de proceso, que se ha usado insistentemente en este texto, se puede analizar desde dos perspectivas: una en la que el proceso ejerce una propiedad sobre los recursos

y otra que se asocia a la ejecución. Tomando en cuenta esos puntos de vista, un proceso se puede ver como:

* Una unidad **propietaria de recursos**, es decir que tiene diversos recursos tales como un espacio de memoria, la CPU (en algunos momentos), archivos abiertos, etc.
* Una unidad de planificación y despacho. El planificador trabaja con procesos (ya sean pesados o ligeros) decidiendo en qué momento y hasta cuándo pueden usar el procesador (los planifica); por otra parte, el despachador **despacha** los procesos cuando realiza el cambio de contexto y garantiza la ejecución futura de los procesos que se desalojan.

Básicamente, la planificación y el despacho de procesos se hacen a nivel de hilo. Por ese motivo, la estructura de datos asociada a los hilos es la que mantiene la información básica para estas tareas. No obstante, algunas acciones afectan a todos los hilos de un proceso y, por lo tanto, el SO las maneja a ese nivel; por ejemplo, suspender un proceso para hacer un intercambio de memoria (se trata en el capítulo III) da como resultado suspender todos los hilos que son hijos de ese proceso, dado que ellos comparten una memoria común; la terminación de un proceso también provoca la terminación de sus hilos.

Comunicación entre procesos a través de una memoria común

Para que dos o más procesos se puedan comunicar a través de una zona de memoria común, es necesario que ellos declaren la memoria común que les servirá de vía de enlace. A continuación, se retoma "el problema del productor-consumidor" para ilustrar esta forma de comunicación.

Se debe recordar que:

* El productor procesa una cierta información y la deposita en un búfer.
* El consumidor extrae del búfer lo que produce el consumidor.
* El productor no podrá producir cuando el búfer esté lleno.
* El consumidor no podrá consumir cuando el búfer esté vacío.

Solución al problema productor-consumidor

La figura I.18 muestra la idea en forma esquemática.

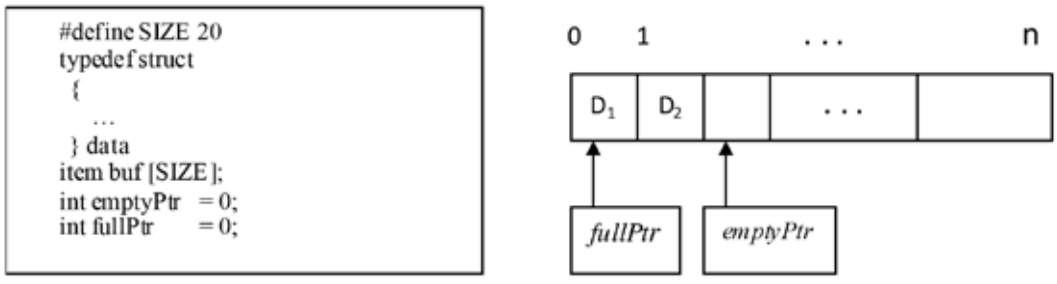


Figura I.18. Productor-consumidor. Búfer acotado. Fuente elaboración propia.

Las variables del cuadro de la izquierda residen en un área de memoria compartida y el búfer (a la derecha) se representa como un arreglo circular que tiene dos apuntadores lógicos:

* La variable emptyPtr apunta a la próxima posición libre del búfer.
* La variable fullPtr apunta a la primera posición llena del búfer.

Otras consideraciones a tomar en cuenta son:

* El búfer está vacío cuando fullPtr == emptyPtr.
* El búfer está lleno cuando ((in + 1) % SIZE) == emptyPtr.

A continuación, se presenta el código para el problema que se ha descrito. La solución mostrada permite que se puedan tener SIZE -1 localizaciones ocupadas a lo sumo, es decir, la posición enésima del búfer siempre estará libre. Más adelante, se presenta una "solución" que trata de ocupar todas las localizaciones, pero antes es necesario hacer un paréntesis para describir someramente algunos aspectos relacionados con el concepto de concurrencia.

```
void producer()
  {
    data Produced;

    while(true)
      {
      //Producir dato, ponerlo en Produced
      while ((emptyPtr + 1) % SIZE == fullPtr)
          ; // no hacer nada
        buf[emptyPtr] = Produced;
        emptyPtr = (emptyPtr + 1) % SIZE;
      }
  }
```

```
void consumer()
  {
    data Consumed;

    while(true)
      {
        while ((emptyPtr == fullPtr)
          ; // no hacer nada
        Consumed = buf[fullPtr];
        fullPtr = (fullPtr + 1) % SIZE;
        // Consumir dato de Consumed
      }
  }
```

1.9.1 Concurrencia

La concurrencia tiene que ver con un conjunto de acontecimientos que ocurren a un mismo tiempo. En el mundo de la computación en general y en los SO en particular, está relacionada con la ocurrencia de dos procesos que actúan en los mismos momentos dentro del SO, pero que se relacionan debido a que comparten algo. En general, en este entorno, la concurrencia se relaciona con:

* La comunicación entre procesos.
* La competencia y el compartimiento de recursos.
* La sincronización de las actividades de múltiples procesos.
* La asignación de tiempo del procesador a los procesos.

El problema del productor-consumidor (segunda versión)

Para tratar de "resolver" la deficiencia señalada antes, de modo que se puedan usar todas las localizaciones del búfer, el código original se transforma y se introduce la variable counter, que se comparte entre ambos procesos. La variable counter se usa para contar los elementos del búfer que están ocupados; cuando counter es igual que 0, el búfer está vacío y cuando es igual que la longitud del búfer (n en el ejemplo de la figura I.17), está lleno.

```
void producer()
  {
    data Produced;

    while(true)
      {
        while (counter == SIZE)
        ;  // no hacer nada
       buf[emptyPtr] = Produced;
       emptyPtr = (emptyPtr + 1) % SIZE;
      counter++;
      }
  }
```

```
void consumer()
  {
    data Consumed;

    while(true)
      {
        while ((counter == 0)
          ;  // no hacer nada
        Consumed = buf[fullPtr];
        fullPtr = (fullPtr + 1) % SIZE;
        counter--;
        // Consumir de Consumed
      }
  }
```

Todo parece estar bien, pero la mala noticia es que, en realidad, si los dos procesos se ejecutan de forma no concurrente, todo estará bien; pero cuando se ejecuten en forma concurrente, los resultados pueden ser erróneos y, aún más, son impredecibles, por tanto, a veces se obtienen resultados correctos y a veces no. Para resaltar esta situación, se presenta el siguiente panorama:

Supóngase que en un instante dado el valor de la variable counter es 10 y que el **productor** y el **consumidor** ejecutan de forma concurrente las sentencias counter++ y counter--, respectivamente. Sorpresivamente para muchos, el valor de la variable counter al final de estas ejecuciones puede ser 9, 10 u 11 cuando el único resultado correcto es 10. ¿Por qué sucede esto?

Ante todo, debe tomarse en cuenta que cualquier sentencia en código fuente de un programa deber ser transformada por el compilador a un código de máquina nativo para poderse ejecutar.

En el caso de las operaciones de adición o sustracción, típicamente, se descompondrán en tres instrucciones de máquina, que se muestran a continuación. En este ejemplo, se usan los registros $register_1$ y $register_2$ de un procesador hipotético:

```
counter++
  register1 = counter
  register1 = register1 + 1
  counter   = register1
```

```
counter--
  register2  = counter
  register2  = register2 - 1
  counter    = register2
```

El problema es que no se sabe cuándo un proceso va a perder o a tomar el control de la CPU, y en ese panorama se pueden presentar situaciones como las siguientes:

1. El productor produce un dato e inicia la actualización de la variable **counter**, logrando hacer las dos primeras operaciones en código nativo, es decir:

$$\text{register}_1 = \text{counter} \quad ; \text{register}_1 \text{ recibe el valor 10}$$
$$\text{register}_1 = \text{register}_1 + 1 \quad ; \text{register}_1 \text{ recibe el valor 11}$$

2. Después de hacer las operaciones anteriores, el SO le quita el procesador al productor. El planificador escoge como próximo proceso a ejecutar al consumidor y le comunica su decisión al despachador.
3. El despachador realiza el cambio de contexto, que consiste en:
   a. Guardar, entre otras cosas, los valores actuales de los registros en el PCB del productor (es su CPU virtual), register_1 queda con el valor 11 (obsérvese que el proceso no llega a actualizar counter, que debería también ser 11).
   b. Tomar los valores del PCB del consumidor y cargar la CPU con esos valores, y darle el control al consumidor que comienza a ejecutar.
4. El consumidor realiza las acciones siguientes (en código nativo):

$$\text{register}_2 = \text{counter} \quad ; \text{register}_2 \text{ recibe 10}$$
$$\text{register}_2 = \text{register}_2 - 1 \quad ; \text{register}_2 \text{ recibe 9}$$
$$\text{counter} = \text{register}_2 \quad ; \text{counter recibe 9}$$

   Obsérvese que el consumidor toma un valor de counter que no es correcto (porque el productor no lo actualizó):

5. Después de hacer la operación anterior, se le retira el procesador al consumidor y se lo da al productor, el cual finaliza actualizando la variable counter (counter = register_1) para informar que hay un nuevo dato que consumir, lo cual da por resultado el valor 11 que es erróneo, dado que se produjo uno y se consumió uno, lo que debería dejar las cosas como estaban al principio, es decir, hay 10 datos y no 11.

Todo el problema que se ha presentado viene dado por el hecho de que la variable counter se comparte entre ambos procesos y no se pueden efectuar operaciones sobre ella sin tomar ciertos cuidados. Se dice que counter es una **variable crítica** porque se comparte entre ambos procesos.

1.9.1 Principios de concurrencia

En los SO multiprogramados, que se instalan sobre sistemas de cómputo con un solo procesador, se tiene la sensación de que varios procesos se ejecutan simultáneamente, pero en realidad lo que se hace es multiplexar el procesador compartiendo el tiempo entre todos los procesos.

De otra parte, los SO de multiprocesamiento, que se deben instalar sobre sistemas de cómputo con varios procesadores, permiten que los procesos realmente se puedan ejecutar a la vez. Si el sistema de cómputo posee **n** procesadores, se podrán ejecutar **n** procesos en forma paralela, es decir, un proceso en cada procesador. Obsérvese que, en este caso, sobre cada procesador se pueden ejecutar también varios procesos.

Ambas técnicas se pueden ver como ejemplos de procesamiento concurrente y presentan los mismos problemas, que en general son:

* Los recursos globales compartidos pueden causar diversos problemas si no se usan en forma adecuada.
* Al SO le resulta difícil manipular óptimamente la asignación de recursos.
* Es difícil localizar un error de programación, sobre todo porque las situaciones en que se ejecutan los programas, por lo general, no son reproducibles.

El problema del productor-consumidor podría resolverse si el acceso al código que hace operaciones sobre la variable común counter (la **variable crítica**) se hubiera hecho en forma excluyente (cuando un proceso A lo esté ejecutando, ningún otro proceso B lo puede hacer). El segmento de código conflictivo recibe el nombre de **sección crítica**; en este caso son críticos:

```
counter++; //en el productor        counter--; //en el consumidor
```

El mismo problema se presenta en un medio de multiprocesamiento, dado que varios procesos ejecutando en el mismo procesador y compartiendo variables dan el mismo resultado erróneo si no se toman cuidados especiales. Pero, además, si dos procesos ejecutan los procedimientos realmente en paralelo (porque lo hacen en procesadores distintos), también se repite la situación. En conclusión, a las variables compartidas hay que usarlas en forma excluyente.

Condiciones de Bernstein

Para que dos secuencias de sentencias, S_i, S_j, se puedan ejecutar concurrentemente, se deben cumplir las siguientes condiciones:

- $I(Si) \cap O(Sj) = \emptyset$
- $I(Sj) \cap O(Si) = \emptyset$
- $O(Si) \cap O(Sj) = \emptyset$

Siendo:

* $I(S_k)$ el conjunto formado por todas las variables de entrada (*input*), es decir que los valores contenidos en ellas se usan durante la ejecución de la secuencia de instrucciones S_k.
* $O(S_k)$ el conjunto de todas las variables de salida (*output*), es decir que cambian su valor durante la ejecución de una secuencia de instrucciones S_k.

Si se cumplen esas tres condiciones, se dice que los conjuntos no tienen una relación de dependencia entre sus variables. Debe observarse que:

1. Si se incumple la primera condición, no existe garantía del valor que tendrían las variables cuando se esté ejecutando la secuencia de sentencias Si, dado que pueden haber sido (o no haber sido) alteradas por la secuencia de sentencias Sj que se está ejecutando concurrentemente. Es decir, el valor que se obtenga en Si es dependiente de Sj.
2. Si se incumple la segunda condición, no existe garantía del valor que tendrían las variables cuando se esté ejecutando la secuencia de sentencias Sj, dado que pueden haber sido (o no haber sido) alteradas por la secuencia de sentencias Si que se está ejecutando concurrentemente. Es decir, el valor que se obtenga en Sj es dependiente de Si.
3. Por último, es fácil notar que existe una dependencia mutua entre Si y Sj cuando se incumple la tercera condición, ya que en ambas secuencias de sentencias se alteran los valores de las variables.

Queda claro que no existe ningún problema cuando $I(S_i) \cap I(S_j) \neq \emptyset$, debido a que en este caso ambas secuencias de sentencias están usando los valores de las variables pero no las están alterando.

Sección crítica

Para dos o más procesos que comparten variables comunes, la sección crítica está formada por el conjunto de sentencias que violan las condiciones de Bernstein.

* Por supuesto que dos procesos que no comparten nada no tendrán secciones críticas entre ellos.
* Por el contrario, si dos procesos comparten algunas variables, no puede ser que uno lea y otro escriba sobre ellas o que ambos escriban sobre ellas sin tomar ningún tipo de cuidado.

Lo importante en este esquema es que las secciones críticas de ambos procesos se ejecuten en forma excluyente; es decir, de entre varios procesos que cooperan entre sí de forma concurrente solo uno de ellos puede estar en cada instante de tiempo dentro de su sección crítica correspondiente.

Para resolver ese problema, primero deben identificarse los segmentos de códigos críticos y escribir un esquema que permita **pedir permiso** para entrar en cada uno de ellos y **avisar cuando sale.**

Obsérvese que todo proceso que pida permiso para entrar en una sección crítica y se le niegue (porque otro con el cual comparte las variables críticas está dentro de su correspondiente sección conflictiva), queda **bloqueado** en espera de un recurso, el cual será liberado cuando el proceso que lo ocupe **avise** que salió de la sección crítica.

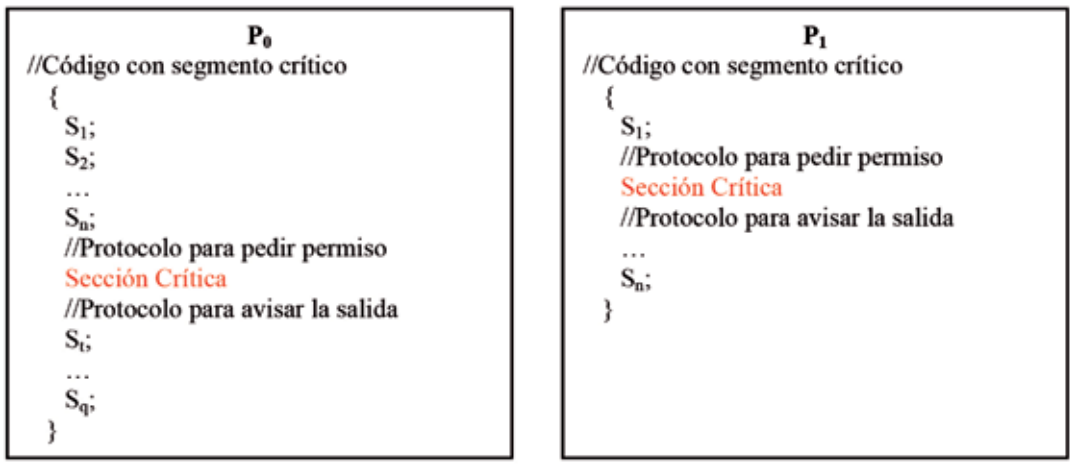


Figura I.19. Esquema general para tratar la sección crítica. Fuente elaboración propia.

La estructura general de esta solución se aprecia en la figura I.19. Debe observarse que antes y después de la correspondiente sección crítica de ambos procesos (en este caso P_0 y P_1) debe establecerse algún protocolo de permiso y aviso para poder entrar con garantía de exclusividad en la parte conflictiva.

Antes de analizar las soluciones a la sección crítica, el lector (como programador) debe tener cuidado y analizar la longitud de los segmentos de programas que son críticos, debido a que es una mala estrategia hacer programas concurrentes con secciones críticas grandes que consumirán mucho tiempo de ejecución mientras se tienen bloqueados los restantes procesos que piden permiso para hacer sus respectivas secciones críticas.

Una solución al problema de la sección crítica para un conjunto $P = \{p_1, p_2, \ldots, p_n\}$ de procesos que declaran secciones críticas entre ellos debe satisfacer los siguientes requisitos:

1. Exclusión mutua. Solo un proceso pi, cualquiera del conjunto P, puede estar ejecutando su sección crítica SCi. Durante este tiempo si algún otro proceso pj $j \neq i$ desea ejecutar su sección crítica SCj, tendrá que esperar a que pi termine.

2. Progreso. Solo los procesos interesados en entrar en la sección crítica pueden participar en la decisión de cuál será el que comenzará a hacerlo, los demás no participan en esa decisión. Esto lleva implícito, por ejemplo, que no se pueden establecer turnos.

3. Espera limitada. Después de que un proceso pi haya hecho una solicitud para entrar en su sección crítica SCi, debe existir un límite de veces que se les permita a otros procesos hacerlo. Esto significa que ningún proceso debe esperar indefinidamente para entrar a su sección crítica, es decir que no puede existir una espera indefinida por razones de prioridad o cualquier otra consideración.

I.9.2 Soluciones por *software*

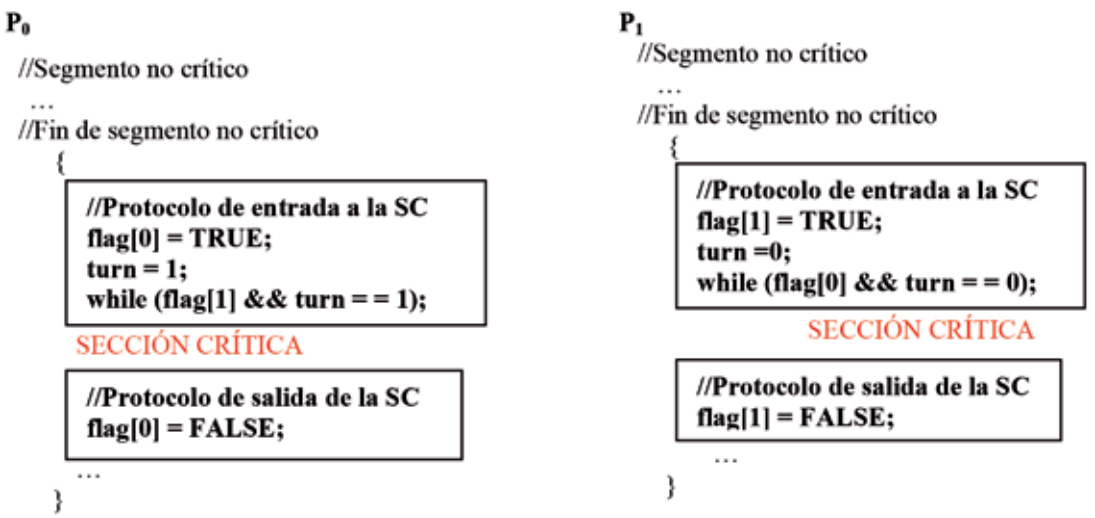
Solución de Peterson

La solución de Peterson al problema de la sección crítica es una solución por *software*. Las soluciones por *software* pueden resultar bastante complicadas, debido a que las variables que participan en los protocolos de entrada y salida también se vuelven críticas, a no ser que se tomen cuidados especiales (ya se vio lo que sucedió con la variable counter en el problema del productor-consumidor).

La solución de Peterson funciona bien en arquitecturas de computadoras más antiguas, pero no en algunas modernas. El problema tiene que ver con la forma en que esas arquitecturas actuales resuelven las operaciones básicas de lenguaje de máquina. No obstante, esta solución es muy clara y se usará en este apartado como un recurso pedagógico.

La solución se restringe a dos procesos (P_0 y P_1), que alternan la ejecución entre su sección crítica y el resto del código y que comparten dos estructuras de datos: turn, que es una variable entera y flag, que es un arreglo booleano de dos elementos.

La variable entera turn se usa para hacer una doble verificación cuando ambos procesos llegan al protocolo de entrada a la vez.



El arreglo flag se usa para indicar cuando un proceso está interesado en entrar en su sección crítica; por ejemplo, si flag[0] es TRUE, entonces el proceso P_0 quiere entrar en su sección crítica. El arreglo flag se inicializa en FALSE, porque inicialmente ningún proceso está interesado en entrar a su sección crítica.

La solución anterior cumple las tres exigencias que debe satisfacer una solución al problema de la sección crítica. A continuación, se muestra el análisis de ellas:

1. Para probar la exclusión mutua, se puede observar que cada proceso entra en su sección crítica solo en las situaciones siguientes:

a. Primera situación, un solo proceso está interesado en entrar a la sección crítica:

   i. Si flag[1] es FALSE, entonces el proceso P_1 no está intentando entrar en la sección crítica y, por tanto, P_0 podría entrar si lo deseara porque su ciclo while se evaluaría como falso.

   ii. Si flag[0] es FALSE, entonces el proceso P_0 no está intentando entrar en la sección crítica y, por tanto, P_1 podría entrar si lo deseara por la misma razón anterior.

b. Segunda situación, ambos procesos están interesados en entrar a sus secciones críticas y, por tanto, el proceso P_0 le asigna TRUE a flag[0] mientras que el proceso P_1 le asigna TRUE a flag[1]. Después, ambos procesos le asignan un valor a turn, pero quedará el último (0 o 1) porque una variable simple solo puede almacenar un valor.

   Luego, cada proceso ejecuta su ciclo while, la condicional del ciclo será verdadera en el proceso que le haya dado el valor final a turn y, por ese motivo, dicho proceso se quedará en un ciclo de espera, mientras el otro proceso entra en su sección crítica porque la evaluación de la condición asociada a la sentencia while da falso.

2. Para probar el progreso, basta con observar que los procesos que no están listos para entrar a la sección crítica tienen la variable flag[i] (i = 0, 1) con el valor FALSE y, por tanto, no toman parte en la decisión de quién entra.

3. La espera limitada se ve claramente al observar que son solo dos procesos, y si ambos piden permiso a la vez (situación 1b), el que no entre quedará probando el ciclo while hasta que el que entró cambie el valor de la variable flag y en ese momento le corresponderá su turno.

I.9.3 Sincronización por *hardware*

El problema de la sección crítica se puede resolver en un sistema de cómputo con un solo procesador si se deshabilitan las interrupciones durante el tiempo en que se usen las variables compartidas, de modo que se ejecuten las instrucciones de forma ininterrumpida; sin embargo, esa solución no es posible en los sistemas de multiprocesamiento y hoy en día cualquier equipo casero tiene un procesador multinúcleo.

Muchos sistemas de cómputo modernos proporcionan instrucciones especiales de *hardware* que se ejecutan de forma atómica. Las instrucciones que se ejecutan en forma atómica no pueden ser interrumpidas y solo un proceso las puede estar ejecutando en cada instante de tiempo.

Algunas de las implementaciones que toman en cuenta la idea de funciones atómicas por *hardware* se discuten a continuación.

Función de *hardware* TestAndSet

La función TestAndSet actúa sobre una localización de memoria y se ejecuta en forma atómica, es decir que si dos procesos intentan ejecutarla, aun cuando estén haciéndolo

desde distintos procesadores, lo harán en forma secuencial, esto es, uno después del otro. El código para esta función es el siguiente:

```
boolean TestAndSet (boolean *target)
  {
    boolean  rv = *target;
    target       = TRUE;
    return rv;
  }
```

Obsérvese que la función recibe un valor, a través de la dirección apuntada por la variable target, que se pasa como único parámetro. Ese valor se asigna a la variable local rv que se retorna con la sentencia return rv, es decir que la función devuelve el valor que recibe a través de target. Dentro de la función se cambia a TRUE el contenido de la dirección apuntada por target.

Si varios procesos intentan hacer la misma operación, solo uno obtendrá acceso (es una operación atómica). La CPU puede usar las instrucciones TestAndSet que ofrecen otros componentes electrónicos o usar las suyas propias.

Lo más importante de la sentencia anterior es que se hace de forma atómica (una vez comenzada no se interrumpe) y que, aun en el caso de que se intente ejecutar desde varios procesadores, el resultado será la ejecución en secuencia, o sea, una después de la otra en un orden indeterminado.

Para resolver el problema de la exclusión mutua, se puede usar la función TestAndSet, declarando una variable global booleana, que denominaremos closed; dicha variable se inicializa a FALSE.

Cada proceso que vaya a usar su sección crítica deberá pedir permiso, usando TestAndSet, antes de entrar y cambiar la variable global closed al salir de ella, de modo que ese cambio incida en el valor devuelto por TestAndSet y permita que otro proceso que haga la prueba pueda entrar una vez que él la abandone.

A continuación, se muestra el uso para dos procesos; observe la explicación que sigue.

```
boolean closed = FALSE;
```

P_i	P_j

{
…
…
while (TestAndSet(&closed); //¿puedo?
SECCIÓN CRÍTICA
closed = FALSE //salí
…
}
	{
	…
	while(TestAndSet(&closed); //¿puedo?
	SECCIÓN CRÍTICA
	closed = FALSE //salí
	…
	}

El código anterior, en general, sigue las ideas siguientes:

1. Se declara una variable booleana **global** a todos los procesos que tienen sección crítica y se inicializa con el valor FALSE, lo que significa que en ese momento no hay ningún proceso dentro de su sección crítica. La variable usada en este caso es closed (puede ser cualquier otra).
2. Cada proceso deberá poner el código siguiente antes de su sección crítica:

```
while (TestAndSet(&closed)); // Si es verdadero, repite el ciclo y no entra
```

   Ese código es la petición de permiso para entrar a la sección crítica y tiene la siguiente funcionalidad:

   - Si varios procesos hacen la llamada a la función TestAndSet en la forma TestAndSet(&closed), entrarán en dicha función en un orden indeterminado de forma secuencial y solo uno a la vez. El primer proceso que ejecute la función obtendrá de retorno el valor FALSE (closed es FALSE) y por eso la condición de while también será FALSE, dando como resultado que ese proceso entre en su sección crítica.
   - Esa ejecución también causa que la variable closed reciba el valor TRUE, debido a que la sentencia target = TRUE dentro de la función TestAndSet hace que se escriba en la dirección apuntada por closed.
   - Los demás procesos que ejecuten TestAndSet (en secuencia) se quedan en una espera activa, probando constantemente la condición de while, que permanecerá con el valor TRUE hasta que el proceso que entró en la sección crítica cambie ese valor.
3. Cada proceso deberá poner el código siguiente después de la sección crítica:

```
closed = FALSE;
```

   - Ese código cambia el valor de la variable global closed a FALSE, lo que causa que el siguiente proceso que ejecute TestAndSet obtenga FALSE como valor de retorno de la función y entre a su sección crítica, pero antes de eso TestAndSet pone la variable closed en TRUE (para bloquear los demás procesos).

Obsérvese que cada proceso tiene su propia sección crítica, en el proceso P_i está más "adentro" del código del programa (observe los puntos suspensivos). Esta aclaración es pertinente porque no se puede asumir nada en relación con la velocidad de los procesos ni con la posición del código dentro del programa.

Semáforos

La solución de *hardware* TestAndSet y otras (como swap, que no se discute en este libro) resultan un poco complicadas para los programadores. Una solución más simple, externamente, es la conocida como **semáforo** (por su parecido con esos controladores

del tránsito). La idea de los semáforos para sincronizar procesos fue concebida por el científico holandés Edsger Dijkstra y se usó por primera vez en el SO THEOS.

Un semáforo **S** se define como una variable entera sobre la cual solo se pueden ejecutar dos acciones: inicializarla y accederla a través de las operaciones atómicas wait() y signal()[27].

Las operaciones wait() y signal() se ejecutan en forma atómica. La definición de ambas operaciones se aprecia seguidamente:

```
wait(S)
  {
    while (S <= 0);
    S--;
  }
```

```
signal(S)
  {
    S++;
  }
```

A continuación, se muestra el código para dos procesos P_i y P_j que usan semáforos para trabajar con secciones críticas.

```
semaphore S = 1;
```

```
Pi
  {
    …
    …
    wait(S)
    SECCIÓN CRÍTICA
    signal(S);
}
```

```
Pj
  {
    wait(S)
    SECCIÓN CRÍTICA
    signal(S);
    …
    …
  }
```

Se declara una variable común, **S**, de tipo semaphore, y se inicializa con el valor 1. El primer proceso que llegue a la operación wait() verifica el valor de S; como no es menor o igual que cero, le resta 1; ahora S tiene el valor cero y el proceso entra en su sección crítica. Cualquier otro proceso al que se le permita hacer la operación wait() quedará bloqueado, debido a que wait() es atómica y solo un proceso puede hacerla en un instante de tiempo dado. El cambio de valor de la variable S lo deberá realizar el proceso que entró a la sección crítica y lo hará usando la operación signal(), cuando finalice su sección crítica.

Tipos de semáforos

Los semáforos pueden ser de dos tipos:

* Binarios. Solo admiten los valores 0 y 1, se conocen también como candados mutuos (el que se usó en el ejemplo precedente es un semáforo binario).
* Contadores. Admiten cualquier valor entero.

27 Las operaciones se nombraron originalmente P (*prolaag*-tratar de reducir) y V (*verhogen*- incrementar).

Los semáforos binarios se pueden usar para tratar el problema de la sección crítica, mientras que los semáforos contadores se pueden usar, además, para controlar el acceso a recursos que tienen una cantidad finita de instancias. Los semáforos también se pueden usar para resolver diversos problemas de sincronización.

La principal desventaja de todas las soluciones presentadas hasta ahora se da por el hecho de que la petición de acceso a la sección crítica causa una espera ocupada, lo cual significa que los procesos a los que se les niega el acceso no se bloquean quedándose inactivos, sino que prueban constantemente la condición de entrada, lo que da por resultado que se malgaste el tiempo del procesador.

Semáforo sin espera ocupada

En esta sección, se describe una implementación de la idea de los semáforos, pero sin espera ocupada. Para implementar esta solución, se definirá una estructura de datos que contiene dos campos:

* El primero es el valor del semáforo en sí (la variable entera que se había explicado antes).
* El segundo campo hace referencia a todos los procesos a los que se les niega el acceso y se bloquean esperando por el recurso (en realidad, la espera es por el semáforo que controla el recurso si se quiere ser más preciso).

```
typedef struct
  {
    int green;
    struct pcb * list;
  } semaphore;
```

Los campos de la estructura semaphore tienen los siguientes significados:

- El campo green contiene el valor del semáforo.
- El campo list es un puntero a una lista de los PCB (Bloque de Control de Procesos) de los procesos que están bloqueados en espera del semáforo.
  - La operación wait(), efectuada por un proceso P_i sobre un semáforo S cuando no se le permita la entrada, bloqueará a P_i y lo agregará a la cola apuntada por list, para lo cual se invoca la llamada al sistema block().
  - La operación signal(), efectuada por un proceso P_j sobre un semáforo S, quitará un proceso P_j ($i \neq j$) de la cola apuntada por list y lo despertará, para lo cual se invoca la llamada al sistema wakeup(). Debe observarse que el hecho de despertarlo solo significa que pasa a la cola de listos y no necesariamente que comience a ejecutar (deberá esperar a que el planificador de periodo corto lo escoja).

Con estas ideas en mente, las definiciones de wait() y signal() (en un metacódigo) pueden expresarse de la forma siguiente:

```
wait(semaphore * S)
  {
    S-->green--;
    if ((S-->green) < 0)  // ¿luz roja?
      {
        add();            // agregar proceso a S-->list
        block();          // bloquear el proceso
      }
  }
```

```
signal(semaphore * S)
  {
    S-->green++;
    if ((S-->v) <= 0)     // ¿luz verde?"
      {
        remove(P);        // sacar un Pi de S-->list
        wakeup(P);        // despertar a Pi,
      }
  }
```

En esta implementación, el valor absoluto del campo green de la estructura S permite conocer la cantidad de procesos que están bloqueados en espera del semáforo.

Las operaciones sobre los semáforos se realizan de forma atómica (obsérvese que el semáforo se corresponde con la variable **S** en este caso).

Debe garantizarse que dos procesos no puedan hacer las operaciones wait() y signal() sobre el mismo semáforo a la vez.

* En un sistema de monoprocesamiento (atiende un solo procesador), se puede lograr ese efecto deshabilitando las interrupciones durante el tiempo que duren esas dos operaciones, lo que no permitirá que se intercalen operaciones de procesos diferentes.
* En un sistema de multiprocesamiento (atiende varios procesadores a la vez), será necesario deshabilitar las interrupciones en todos los procesadores durante el tiempo que duren esas dos operaciones, lo cual trae serios inconvenientes y baja considerablemente el rendimiento del sistema.

I.9.4 Una solución al problema del productor-consumidor

Al principio de la sección I.9, se presentó el problema del productor-consumidor, el cual se tomó como ejemplo motivador para insistir en lo sutiles que muchas veces resultan los problemas de sincronización entre procesos; así como las graves consecuencias que traen algunas "soluciones", como la que se dio en aquel momento para tratar de mantener el búfer lleno. La figura I.20 presenta una solución a esa problemática haciendo uso de los semáforos.

La solución hace uso de los dos tipos de semáforos: binario, declarado como mutex semaphore y contador, declarado como count semaphore.

* El semáforo binario lock se usa para controlar el acceso exclusivo al búfer acotado (tiene m entradas).
* El semáforo contador hManyEmpty cuenta la cantidad de espacios libres.
* El semáforo contador hManyFull cuenta la cantidad de espacios ocupados.

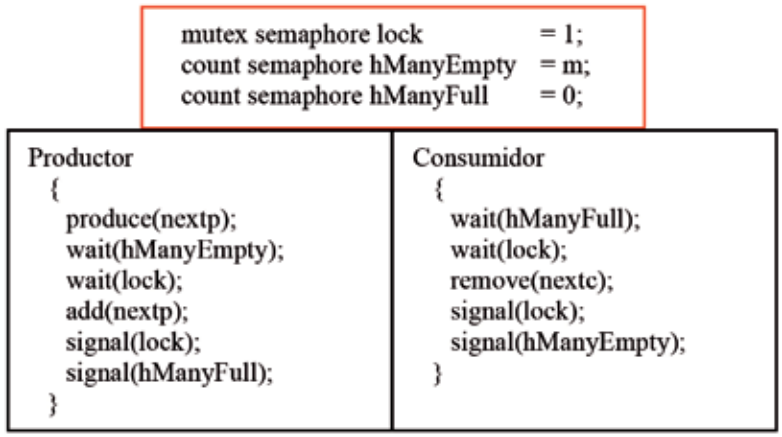


Figura I.20. El problema del productor-consumidor. Fuente: Adaptado de Operating System Concepts.

Inicialmente, el búfer está vacío y por eso a la variable hManyEmpty se le asigna m que es la longitud del búfer.

El productor realiza su trabajo de la siguiente forma:

1. Primero produce algo: produce(nextp)
2. Después, verifica si hay espacio en el búfer para poner lo producido: wait(hManyEmpty), esta llamada al sistema verifica si hManyEmpty > 0. Si la condición anterior se cumple, entonces existe al menos un espacio que se tomará restándole 1 a la variable hManyEmpty, lo cual indica que hay un espacio menos.
3. A continuación, garantiza que el otro proceso no esté accediendo al búfer, y lo hace usando la llamada al sistema wait(lock).
4. Con las dos condiciones anteriores verificadas, se puede poner en el búfer el elemento nextp que se ha producido, para lo cual usa la función add(nextp).
5. Una vez que ha depositado la nueva producción, avisa que salió de la sección crítica, para lo cual hace la llamada al sistema signal(lock) e informa que hay un nuevo elemento, incrementando en uno el valor de la variable hManyFull a través de la llamada al sistema signal(hManyFull).

El trabajo del consumidor es muy parecido al del productor:

* Tiene que pedir permiso para entrar a la sección crítica (wait(lock)) y avisar que salió (signal(lock)).
* Además, en lugar de comprobar si hay espacio antes de entrar, verifica que haya algo que consumir, lo hace usando la llamada al sistema wait(hManyFull). En la salida, deberá informar que hay un nuevo espacio vacío, usando la llamada al sistema signal(hManyEmpty).

Se puede apreciar que existe una relación directa entre ambos procesos, y de hecho, cada uno produce para el otro: el productor produce entradas llenas para el consumidor, mientras que el consumidor produce entradas vacías para el productor. Este tipo de relación es típico de muchos sistemas, por ejemplo, en bases de datos, en aplicaciones cliente-servidor, etc.

I.9. RESUMEN DEL CAPÍTULO

Existe una sutil pero importante diferencia entre proceso y programa; mientras el primero es un **ente activo**, el segundo es un **ente pasivo** que no hace nada y simplemente reside sobre algún medio, el cual no tiene que estar asociado a una computadora.

Un **proceso** es un **programa en ejecución** que posee un conjunto de recursos asignados (de *software* y de *hardware*) y es también **una unidad de planificación y despacho**.

El módulo, que tiene la tarea de planificar el uso de la unidad de procesamiento central, usa diferentes algoritmos para planificar cuándo se le debe asignar ese importante recurso a un proceso dado. Los algoritmos que se usan pueden ser **con desalojo** o **sin desalojo**; en el primer caso, el **planificador le retira** la CPU al proceso en el momento que estime conveniente (en realidad, de acuerdo con el algoritmo), sin tomar en cuenta si el proceso ha terminado o no; en el segundo caso, los procesos **abandonan la CPU** cuando tienen que realizar tareas para las cuales no necesitan el procesador.

Los procesos están representados dentro del SO por la estructura de datos conocida como **Bloque de Control de Proceso** (PCB), la cual funciona como la **CPU virtual del proceso** y permite que los procesos entren y salgan del estado de ejecución, conservando lo que han hecho a pesar de ser interrumpidos constantemente.

La vida de los procesos transcurre por **diversos estados** que describen su actividad, entre los que se destacan los estados de: listo, ejecutando, terminado, bloqueado, etc.

Cuando dos o más procesos comparten recursos, que se deben usar en forma exclusiva, pueden ejecutar de forma concurrente si se tienen cuidados especiales en sus **secciones críticas**.

I.10. EJERCICIOS

1. Explique las diferencias entre planificación con desalojo y sin desalojo.
2. Dados los datos de los procesos relacionados en la tabla siguiente y tomando en cuenta que no existe el desalojo:

LLEGADA DE PROCESOS A LA COLA DE LISTOS		
PROCESO	**TIEMPO DE LLEGADA**	**TIEMPO DE CPU**
P_1	0.0	6
P_2	0.4	3
P_3	1.0	9

a. Calcule el promedio de tiempo para los procesos si se usan los algoritmos de planificación FCFS, SJF y Round Robin.

   Nota. Recuerde que el tiempo para un proceso se mide desde que el proceso entra al sistema hasta que termina.

3. Los procesos que están ejecutándose pasan su vida en dos etapas fundamentales. ¿Cuáles son y qué significa cada una?

   a. ¿Qué entiende usted por una buena mezcla de trabajo? ¿Cómo la formaría?

4. Las colas multinivel se caracterizan porque los trabajos se asignan a colas fijas y solo se atienden las colas inferiores cuando las superiores están vacías. ¿Qué implicaciones tiene esa forma de planificar sobre los procesos de las colas inferiores? ¿Cómo se resuelve?

   a. Explique, con sus palabras, las diferencias y semejanzas que existen entre las colas multinivel y las colas multinivel con retroalimentación.

5. Explique el papel que juega el Bloque de Control de Procesos (PCB) cuando se hace un cambio de contexto.

6. Presente un ejemplo propio que muestre la importancia de la operación dual.

7. Muestre un ejemplo de la vida cotidiana que se asemeje al problema de la sección crítica.

   a. Explique las condiciones que debe cumplir una solución al problema de la sección crítica.

8. Algunas de las soluciones presentadas para resolver el problema de la sección crítica usan la espera ocupada y otras no. Diga qué significa la espera ocupada. ¿Qué implicaciones tiene su uso?

9. Diga cuáles son los planificadores que pueden existir en un SO.

   a. Explique el rol de cada uno de ellos.

10. Explique la relación que existe entre el planificador de periodo corto y el despachador.

11. ¿Por qué se dice que el PCB es una CPU virtual para los procesos?

12. Explique las semejanzas y diferencias entre proceso pesado y proceso ligero, y justifique el nombre que reciben.

    a. Ofrezca una definición para proceso pesado y otra para proceso ligero en la que queden claras esas diferencias.

13. Diga los momentos en que un proceso puede cambiar de estado y explique por qué se produce ese cambio.

a. Investigue los nombres de los diferentes estados por los que pasa un proceso durante su existencia en los siguientes SO:

i. Windows (cualquier versión).

ii. Unix (cualquier implementación).

iii. Linux (cualquier implementación).

14. Algunos SO pertenecen al movimiento del software libre. Explique las libertades que debe cumplir un *software* para ser de ese movimiento.

a. ¿Cuál es el significado conceptual de la sigla GNU?

b. ¿Qué significa la sigla POSIX y qué implicaciones tiene que un SO siga ese estándar?

15. Existen diversos criterios para agrupar los SO.

a. Explique cuáles son esos criterios.

b. Diga cómo se denominan los SO que cumplen los distintos criterios y ponga ejemplos de SO que se agrupen bajo esos nombres.

16. Analice qué sucede cuando dos procesos actúan sobre una variable común, ¿siempre habrá problemas?, ¿cuándo sí y cuándo no?, ¿por qué? Analice cada situación.

I.11 BIBLIOGRAFÍA CONSULTADA

Abraham, S., Baer- Galvin, P., & Gagne, G. (2013). *Operating system concepts* (9.ª ed.). Nueva Jersey: Jhon Wiley & Sons.

Bernstein, A. J. (1966). Analysis of programs for parallel processing. *IEEE Transactions on Electronic Computers*, *15*(5), 757-763.

Dhamdhere, D. M. (2008). *Operating systems. A concept based approach.* Nueva York: McGraw-Hill Education.

Elmasri, R., Carrick, A., & Levine, D. (2009). *Operating systems: A spiral approach.* Nueva York: McGraw-Hill.

Feautrier, P. (2011). Bernstein's conditions. En D. Padua (ed.), *Encyclopedia of Parallel Computing* (pp. 130-134). Nueva York: Springer.

Hart , J. (2010). *Windows system programming* (4.ª ed.). Reading: Addison-Wesley.

Harvey, M., Deitel, P., & Choffnes, D. (2003). *Operating systems* (3.ª ed.). Nueva Jersey: Prentice Hall.

Oaks, S., & Wong, H. (2004). *Java Threads* (3.ª ed.). Sebastopol: O'Reilly Media.

Peterson, G. (1981). Myths about the mutual exclusion problem. *Information Processing Letters*, *12*(3), 115-116.

Robbins, K., & Robbins, S. (2003). *Unix Systems programming: Communication, concurrency, and threads.* Nueva Jersey: Prentice Hall.

Stallings, W. (2014). *Operating systems: Internals and design principles* (8.ª ed.). Nueva York: Pearson.

Tanenbaum , A. (2006). *Operating systems: Design and implementation* (3.ª ed.). Nueva Jersey: Prentice Hall.

Tanenbaum, A., & Bos, H. (2014). *Modern operating systems* (4.ª ed.). Nueva York: Pearson.

CAPÍTULO II

El sistema de archivos

RESUMEN

Este capítulo se basa en el concepto abstracto de archivo para explicar la forma como se organiza la información sobre los soportes de almacenamiento externos. Se ofrece una visión general de los atributos que se asocian a los archivos, así como las operaciones que se pueden efectuar sobre ellos y la manera como se organizan en directorios jerárquicos. Se presenta la arquitectura general del sistema de archivo y se analiza el concepto abstracto de bloque como unidad mínima de almacenamiento y de manejo de los archivos. Se ofrecen detalles de los sistemas de archivos de colocación o asignación (contigua, enlazada e indexada), analizando las ventajas y desventajas de cada uno de ellos. También se discuten distintos métodos generales para controlar el espacio libre y se explican las particularidades de los sistemas de archivos que están asociados a los sistemas operativos Windows (FAT y NTFS) y a los sistemas operativos tipo UNIX (UFS, ext, ext2, ext3 y ext4). El capítulo finaliza con un resumen y una sección de ejercicios propuestos.

Palabras clave: sistema de archivos, archivos, directorios, bloques, métodos de asignación de espacios, control de espacio libre.

¿Cómo citar este capítulo? / How to cite this chapter?

Lezcano-Brito, M. G. (2017). El sistema de archivos. En *Fundamentos de sistemas operativos. Entornos de trabajo* (pp. 77-104). Bogotá: Ediciones Universidad Cooperativa de Colombia.

CHAPTER II

File system

ABSTRACT

This chapter is based on the abstract concept of file to explain how information is organized on external storage media. It provides an overview of the attributes associated with files, as well as the operations that can be performed on them and how they are organized into hierarchical directories. The general architecture of file systems is presented and the abstract concept of block as the minimum file storage and management unit is analyzed. Details of placement and allocation (contiguous, linked and indexed) file systems are given, evaluating the advantages and disadvantages of each of them. Different general methods to control free space are also discussed and the peculiarities of file systems that are associated with Windows operating systems (FAT and NTFS) and UNIX operating systems (UFS, ext, ext2, ext3 and ext4) are explained. The chapter ends with a summary and a section of proposed exercises.

Keywords: File system, files, directories, blocks, space allocation methods, free space control.

Desde el punto de vista de los usuarios finales, la parte más "visible" de cualquier SO es su **sistema de archivo**. Este fenómeno ocurre debido a que la mayoría de los usuarios hace trabajos que tienen efectos directos en este subsistema y los realizan, además, de forma interactiva.

Se puede añadir que la mayoría de las aplicaciones también actúa con el sistema de archivo si se toma en cuenta que su propósito principal es procesar datos, los cuales muchas veces (en realidad, casi siempre) residen en archivos y el resultado de procesar esos datos se almacena en archivos. El fenómeno es tan común que algunas personas tienen una visión simplificada del SO al reducirlo al sistema de archivo.

El sistema de archivo está formado por dos partes perfectamente distinguibles (al menos en funciones): un **conjunto de archivos** que almacenan los datos en sí y una estructura de datos, conocida como **directorio**, que permite localizar la información contenida en los archivos. Ambos aspectos, entre otros, se tratan en este capítulo.

II.1 EL CONCEPTO DE ARCHIVO

Las computadoras pueden almacenar información permanente en diferentes soportes, tales como discos (de varios tipos), cintas magnéticas, memorias *flash*, etc. La naturaleza de cada uno de estos elementos de almacenamiento es disímil, y para poder trabajar con eficiencia, debe existir una forma homogénea de tratar sus contenidos. El SO hace una abstracción de los elementos físicos que se relacionan con cada uno de los equipos de almacenamiento y trata los datos como una **unidad lógica** que recibe el nombre de **archivo**.

Un archivo es un conjunto de datos que tienen un formato común y se agrupan bajo un nombre. Ese nombre se usa para hacer operaciones sobre el archivo, las cuales pueden ser: borrarlo, copiarlo, moverlo, renombrarlo, leerlo, etc.

Cuando se dice que el conjunto de datos tiene un formato común, se está estableciendo una forma para organizar el archivo, que puede ser una simple cadena de bytes (como lo hace Unix) o un conjunto de registros.

II.2 LA ARQUITECTURA DEL SISTEMA DE ARCHIVO

Una forma típica de organizar el sistema de archivo se presenta en la figura II.1 (puede diferir de acuerdo con las particularidades del SO). En ella, el sistema se presenta como un conjunto de capas: las capas inferiores están más cercanas al *hardware* y las

superiores están más cercanas a los usuarios. Esta organización hace que se alcancen mayores niveles de abstracción en las capas superiores, lo que permite que los usuarios se aíslen de las especificidades y complejidades asociadas a los equipos periféricos.

La capa de los **manipuladores de equipos** (*device driver*) es la que actúa directamente con el *hardware*, es decir, los periféricos de entrada/salida, como los canales o los controladores. La capa es responsable de iniciar la operación de entrada/salida (E/S) sobre el equipo y completarla.

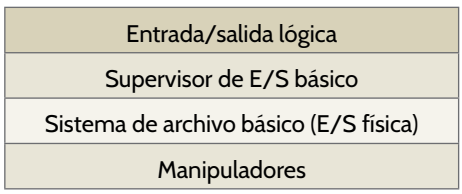


Figura II.1. Organización típica de un sistema de archivos. Fuente elaboración propia.

El **sistema de archivo básico** es el primer enlace entre el sistema de cómputo y el mundo externo. A este nivel, se trabaja con bloques de datos (unidades de información de un cierto tamaño) que se intercambian con el equipo y se pasan desde o hacia la memoria. En esta capa, no se conoce nada acerca de la estructura que tienen los datos que se manipulan.

El **supervisor de E/S básico** es el responsable de comenzar y culminar las entradas y salidas sobre los archivos. Ya en este nivel se usa una estructura de control que permite conocer la situación de los equipos y el estado de los archivos que se almacenan en ellos, entre otros pormenores.

La capa de **entrada/salida lógica** es la responsable de presentarles a los usuarios y a las aplicaciones una interfaz para acceder a la información que se almacena en los equipos. En esta capa, se trabaja con archivos y no con bloques, como lo hace la capa de entrada/salida física; este es el nivel que "ven" los usuarios finales y según su punto de vista este es el único nivel que existe, ya que ellos realizan sus operaciones sobre el concepto abstracto de archivo (copian o mueven archivos, ordenan su ejecución, etc.).

II.3 ATRIBUTOS DE LOS ARCHIVOS. OPERACIONES SOBRE ELLOS

Asociado a todo archivo existe un conjunto de atributos que puede variar de un SO a otro (aunque algunos son invariantes). Entre esos atributos cabe mencionar los siguientes:

* El nombre del archivo. Identifica unívocamente al archivo dentro de la estructura de directorio.
* El tipo de archivo. Se refiere al formato de los archivos, que puede ser de texto, binario, de enlace simbólico, etc.
* Datos acerca de la creación o modificación del archivo, tales como fecha, hora, etc.
* El tamaño del archivo, puede estar especificado en bytes, bloques, etc.
* Atributos de protección. Especifican qué se puede hacer con el archivo y quién puede hacerlo, por ejemplo: leerlo solamente, de lectura/escritura, quién es el dueño, etc.

Como ya se ha apuntado, sobre los archivos se pueden realizar diversas operaciones; la mayoría de esas operaciones necesita buscar información en el **directorio**.

Antes de acceder a la información contenida en un archivo es necesario abrirlo, para lo cual se busca su nombre en los directorios. Una vez localizado el archivo, se pone una referencia a él dentro de una estructura de datos que se denomina **tabla de archivos abiertos**, la cual permite conocer cuál es la próxima entrada del archivo que se leerá en acceso secuencial, quién o quiénes pueden hacer esa operación, etc. La tabla de archivos abiertos permanece siempre en memoria y la referencia citada estará en ella hasta que se cierre el archivo.

Dentro de la tabla de archivos abiertos, los archivos se referencian por números, conocidos como manipuladores (*handles*), o sea, en esa tabla los archivos "pierden" su nombre temporalmente.

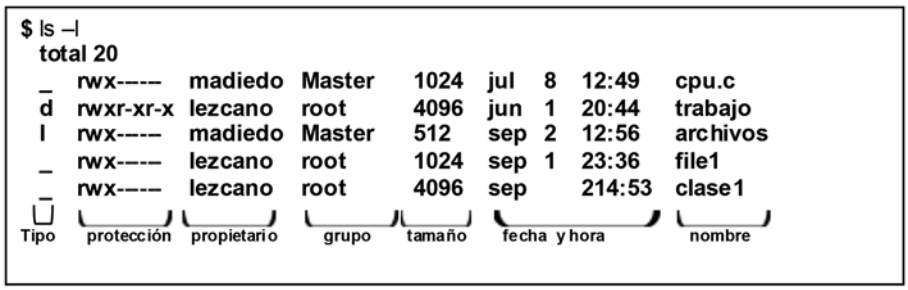


Figura II.2. Algunos de los atributos de los archivos en un SO Unix. Fuente elaboración propia.

La figura II.2 muestra el resultado de ejecutar el comando ls -l[28] en un SO de la familia Unix. Pueden observarse algunos de los atributos de los archivos, tales como: su tamaño, la fecha de modificación, el propietario, etc. Otros atributos, como la referencia al **nodo-i** (se explica más adelante), permanecen ocultos al usuario. En este caso, la primera columna del listado especifica el tipo de archivo de acuerdo con el siguiente convenio:

* **-**, representa un archivo regular.
* **d**, especifica que el archivo es un directorio.
* **l**, especifica un enlace (*link*) simbólico, etc.

La segunda columna es una tira de nueve bits que establece los permisos sobre el archivo. La tira se divide en tres ternas de bits: la primera terna especifica los permisos para el dueño del archivo (*owner*), la segunda establece los permisos para los miembros del grupo (*group*) y la tercera fija los permisos para los otros usuarios (*others*). Si todos los bits de una terna tienen el valor 1, externamente se ven en la forma rwx, donde: r especifica permiso de lectura, w especifica permiso de escritura y x especifica permiso de ejecución; el símbolo - en la posición de un elemento significa que el bit tiene el valor cero y que no se tiene el permiso que esa posición específica.

28 ls -l. Comando de Unix para listar (*list*) en formato largo (*long*).

II.4 ESTRUCTURA DE DIRECTORIO

Todo sistema de archivo posee, como parte de su organización, una estructura de datos denominada **tabla de directorio** (puede que tenga otro nombre pero con la misma funcionalidad), que sirve para localizar los archivos. Todos los medios de almacenamiento poseen esa estructura de datos, que se construye cuando se le da formato al soporte.

La forma que tenga el directorio depende del SO; la figura II.3 presenta una tabla de directorio típica de un SO hipotético. Como se puede apreciar, en esta estructura de datos están presentes diversos campos para especificar datos acerca de los archivos (el encabezado de la primera fila de la tabla solo tiene un fin didáctico, pero no forma parte de ella).

NOMBRE	TIPO	DIRECCIÓN DE INICIO	FECHA	HORA	TAMAÑO	PROPIETARIO

Figura II.3. Tabla de directorio de un SO hipotético. Fuente elaboración propia.

Algunos sistemas de archivos no poseen mucha información en el directorio. Se pueden tener dos extremos: en uno el directorio solo contiene un nombre y un puntero a otra estructura que complementa la información (por ejemplo, Unix); en el otro extremo, el directorio contiene toda la información relativa al archivo, incluso su localización completa (el ya obsoleto CP/M).

La estructura del directorio puede ser de un solo nivel (casi no se usa hoy en día) o de varios niveles o jerárquica. La figura II.4 muestra la estructura de directorio típica de un SO de la familia Unix.

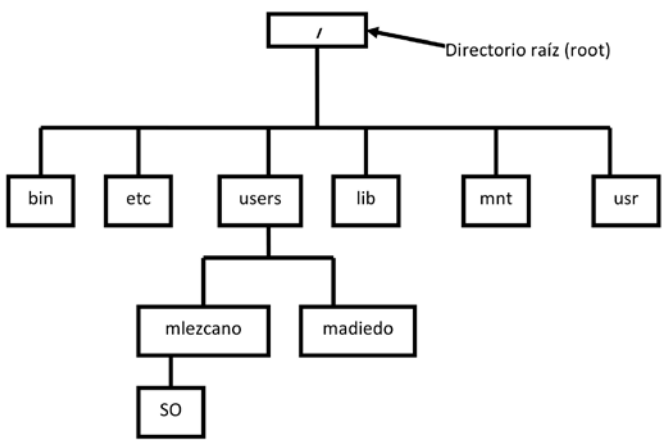


Figura II.4. Estructura de directorio típica de Unix. Fuente elaboración propia.

Desde el punto de vista de los usuarios y las aplicaciones, existen dos formas de acceder a la información contenida en una estructura como la mostrada en la figura II.4:

* Camino o ruta absoluta

  Especifica el camino o la ruta que conduce hasta el directorio o archivo que se desea localizar, tomando como punto de partida el directorio raíz. Por ejemplo, los caminos: /users/mlezcano y /users/madiedo trazan la ruta desde el directorio raíz (se simboliza con el signo / al inicio de la cadena) hasta los directorios mlezcano y madiedo, respectivamente.

* Camino o ruta relativa

  Especifica el camino que conduce hasta el directorio o archivo que se desea localizar a partir de algún directorio diferente al directorio raíz. Por ejemplo, el camino mlezcano/SO que comienza en el directorio mlezcano.

II.5 EL CONCEPTO DE BLOQUE

Un **bloque es la unidad mínima de asignación de espacio** en un soporte específico. Esto quiere decir que, siempre que se pida un cierto espacio en un soporte de almacenamiento, el sistema de archivo proporciona una cantidad que, por lo regular, es mayor que la que se pide.

El problema de asignar el espacio por bloques viene dado por el hecho de que sería muy poco práctico que cada byte del sistema de archivo fuera accedido de forma individual, dado que se gastaría más espacio para controlarlo que para guardar información útil.

Esta forma de asignar espacio hace que se pierdan algunos bytes, a veces todos menos uno, en el último bloque del archivo. Este fenómeno se conoce como **fragmentación interna** y no tiene solución.

El problema de la fragmentación interna en los archivos es fácil de apreciar. Con tal propósito, realice el siguiente ejercicio:

Cree un archivo (en un ambiente Windows) como se muestra en la figura II.5. El archivo se crea con un solo carácter (la letra A), pues ni siquiera se ha añadido cambio de línea después de ese carácter (^z es decir, control Z, es el fin de archivo en los SO de la familia Windows). Después, se usa el comando dir para que muestre el tamaño del archivo que, como es lógico, tiene un solo byte.

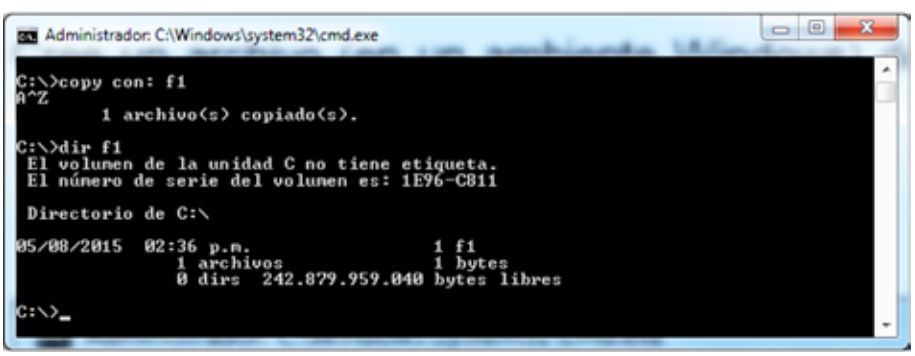


Figura II.5. Creación de archivo desde el intérprete de comandos de Windows. Fuente: Captado desde el SO Windows.

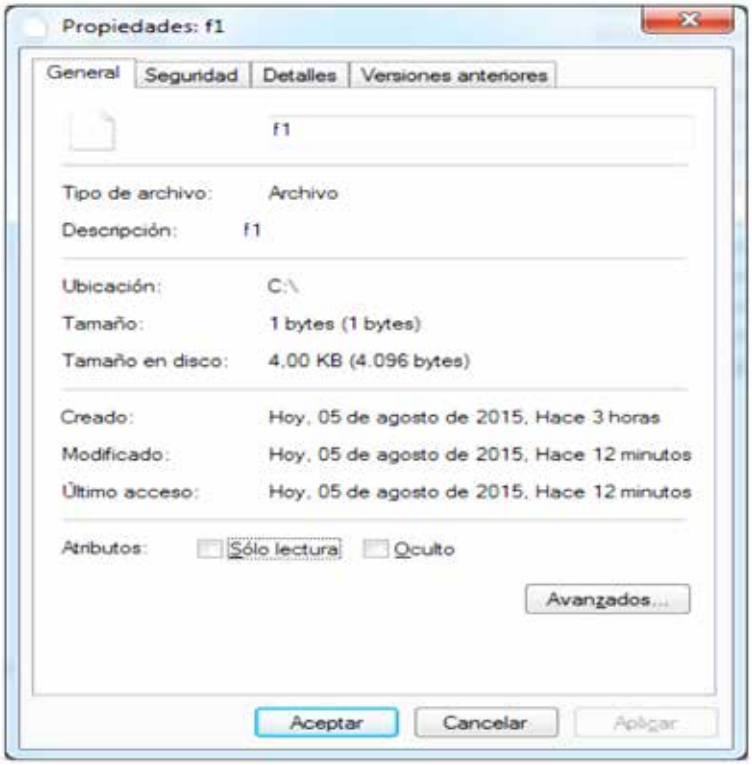


Figura II.6. Muestra de fragmentación interna. Fuente: Captado desde el SO Windows.

Ahora vaya al ambiente gráfico y oprima el botón derecho del ratón encima del nombre del archivo f1. Observe la figura II.6, que debe ser similar a la que usted obtenga; puede notarse que el tamaño del archivo es un byte pero en el disco ocupa 4096 bytes, lo cual viene dado porque el tamaño de los bloques en ese disco es de 4096 bytes y el SO no puede asignar espacio menor que un bloque; por tanto, de los 4096 bytes asignados al archivo f1, solo se utiliza un byte y el resto, aunque pertenece a él, no lo necesita (al menos por el momento) provocando una fragmentación interna de 4095 bytes.

En otra unidad, la fragmentación podría ser mayor o menor porque cada una tendrá un tamaño de bloque que normalmente es mayor mientras mayor sea su capacidad, aunque ese tamaño se puede cambiar cuando se formatea la unidad. El comando chkdsk, que se usa para hacer verificaciones acerca de la integridad de la unidad, también reporta el tamaño de los bloques.

La información acerca del tamaño de los bloques, entre otras, está contenida en una parte del soporte de información que el SO reserva para su trabajo.

II.6 ASIGNACIÓN DE ESPACIO

Existen diferentes métodos y formas para asignar el espacio en un soporte de almacenamiento externo. Para considerar estas formas se deben tomar en cuenta los siguientes criterios:

1. Cuando se cree un nuevo archivo, ¿se le asignará todo el espacio que necesita?
2. ¿Qué tamaño debe tener un bloque?

Si a la primera pregunta se responde que sí, habrá que asignarle todo el espacio que necesitará el archivo cuando se cree, pero es difícil (e incluso imposible) conocer esa información en ese momento, lo cual puede derivar en la sobrestimación de necesidades (malgastando espacio). Por ese motivo, es mejor realizar asignaciones dinámicas, lo cual consiste en hacer crecer el archivo pidiendo bloques nuevos cada vez que se necesite más espacio.

Para la segunda pregunta, debe analizarse que:

* Un tamaño de bloque muy pequeño exige disponer de una tabla enorme para controlar los bloques, es decir, para llevar el control de cuáles están ocupados y cuáles están libres.
* Un tamaño muy grande del bloque gasta mucho espacio, dado que el último bloque de cada archivo puede quedar prácticamente vacío.

A partir del análisis anterior, se infiere que hay que tomar una posición intermedia, en la cual los bloques sean lo suficientemente pequeños para no provocar tanta fragmentación interna y lo suficientemente grandes para no tener que usar demasiados recursos para controlarlos. Queda claro que no es tan fácil definir el tamaño "ideal" de un bloque.

II.6.1 Tipos de asignación

Existen tres formas básicas para asignar espacio en equipos de almacenamiento externo. En este apartado, se analiza cada una de ellas destacando sus ventajas y desventajas.

Asignación contigua

Este tipo de estrategia consiste en asignar un conjunto de bloques contiguos en el momento de la creación del archivo, como se aprecia en la figura II.7.

Obsérvese que: el archivo file1 comienza en el bloque 1, mientras que el archivo file2 comienza en el bloque 4, ambos números actúan como una especie de apuntador al primer bloque del archivo; la figura II.7 usa flechas discontinuas para resaltar esa idea.

En los sistemas de asignación contigua, solo se necesita conocer el **bloque de inicio** y la **cantidad de bloques** que ocupa cada archivo, debido a que los bloques son consecutivos y están en orden a partir del primero que se especifica en la tabla de directorio. En el ejemplo, el archivo file1 está formado por tres bloques que son: 1, 2 y 3; mientras que los dos bloques del archivo file2 son el 4 y el 5.

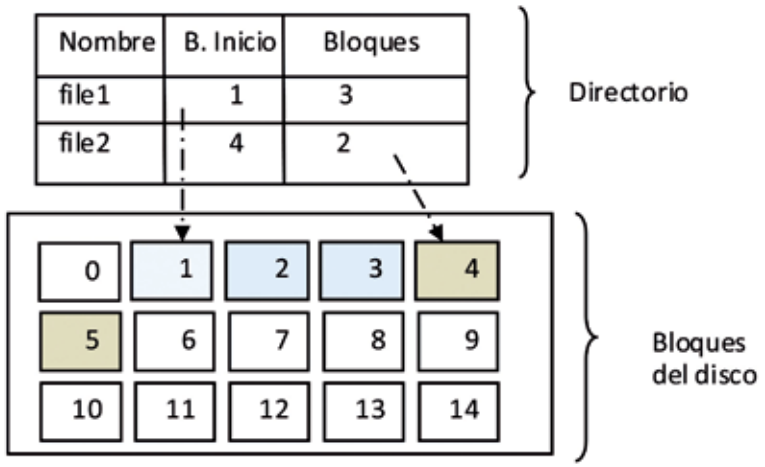


Figura II.7. Asignación contigua. Fuente elaboración propia.

Ventajas:

1. Permite el **acceso directo** a los datos, para lo cual solo se necesita la dirección de inicio del archivo (está contenida en la tabla de directorio) y el desplazamiento a partir de esa dirección.
2. En los discos, el acceso es más rápido debido a que se efectúan menos movimientos mecánicos del cabezal de lectura/escritura para acceder a los datos (están unos a continuación de otros).
3. La recuperación de datos perdidos en algún borrado es más fácil y hay mayores garantías de que sea exitosa.

Desventajas:

1. Un archivo solo puede crecer hasta el inicio de su vecino, ya que cualquier otro "hueco" que esté libre no estará contiguo a él. Este problema hace que los usuarios tiendan a sobrestimar la longitud de los archivos y reserven espacios que en realidad son mayores que los que necesitan, lo que implica un malgasto de espacio. En el ejemplo de la figura II.7, el archivo file1 no puede crecer debido a que está acotado por el archivo file2.
2. Provoca **fragmentación externa**, debido a que puede que no sea posible satisfacer una solicitud de espacio a pesar de existir el espacio en forma de "huecos", o espacios no contiguos, y que sumados satisfacen la petición. La mayoría de los SO que usan este tipo de asignación proveen algún mecanismo de desfragmentación que tiene como fin reunir todos los huecos en uno solo en forma contigua, pero este mecanismo es costoso ya que implica la detención de todos los trabajos que se realizan sobre el periférico.

A partir de este análisis, se concluye que las desventajas de la asignación contigua tienen un peso mayor que sus ventajas, de ahí que no sea muy utilizada actualmente, a no ser en soportes cuya naturaleza permita almacenar grandes archivos por tiempos prolongados con el propósito de resguardarlos.

Estrategias para la asignación de espacio en asignación contigua

Para satisfacer una petición de tamaño m, el sistema de archivo tiene que encontrar un espacio de tamaño n que cumpla que m ≤ n. Existen tres estrategias o políticas para tomar la decisión respecto a cómo escoger ese espacio:

1. El primer acceso. Significa tomar el primer espacio que satisfaga la demanda, es decir que cumpla que m ≤ n.
2. El mejor acceso. Significa tomar un espacio que satisfaga la demanda y que sea el menor de todos los posibles.

   Es decir, m ≤ n y n = menor h_i de H = {h_1, h_2,…}, donde H representa el conjunto de todos los espacios libres.
3. El peor acceso. Significa tomar un espacio que satisfaga la demanda y que sea el mayor de todos los posibles.

   O sea, m ≤ n y n = mayor hi de H = {h1, h2,…}.

Debe observarse que la estrategia del mejor acceso tiende a dejar pequeños espacios dispersos (se dice que el medio está fragmentado). Esos pequeños espacios muchas veces no son útiles a ninguna demanda, de forma que el "mejor acceso" puede resultar siendo la peor solución.

De otra parte, la estrategia del peor acceso deja espacios mayores que, en general, tendrán mayor probabilidad de ser útiles.

En relación con el primer acceso, es totalmente impredecible el resultado ya que dependerá del orden en que estén los espacios libres.

Asignación enlazada

La asignación enlazada trata de resolver los problemas del método anterior. En este esquema, cada bloque contiene un puntero al próximo bloque y los bloques pueden estar en cualquier parte del equipo de almacenamiento.

Observe la idea esquematizada en la figura II.8. Cada bloque tiene dos partes: la primera está destinada a datos y la segunda contiene un puntero. En el caso que se analiza en la figura II.8, el directorio especifica que el primer bloque del archivo file1 es el 2 (los números de los bloques comienzan en cero); el primer campo del bloque 2 contiene el dato d_1 y su puntero apunta al bloque 5 (las líneas discontinuas son para reafirmar la idea); el primer campo del bloque 5 contiene el dato d_2 y su puntero apunta al bloque 7, el cual contiene el dato d_3 y su puntero es nulo, lo que significa que ese es el último bloque del archivo file1 que está compuesto por los bloques 2, 5 y 7.

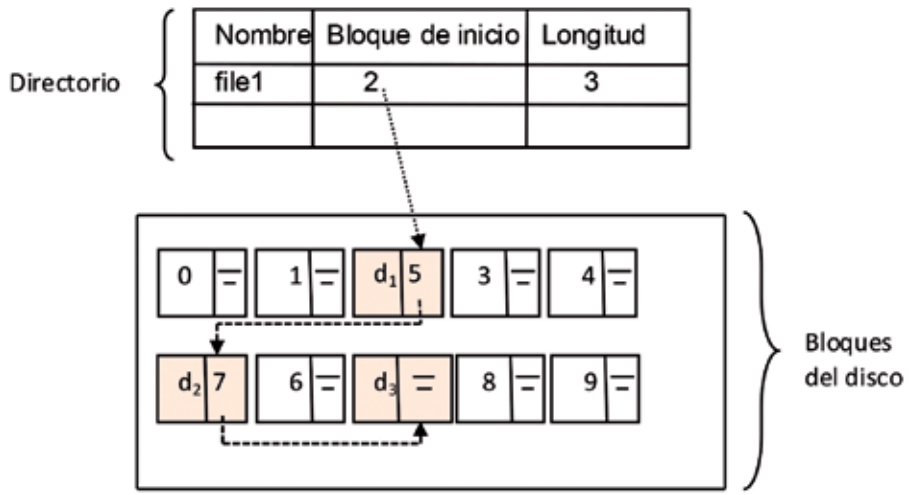


Figura II.8. Asignación enlazada. Fuente elaboración propia.

Ventajas:

1. Como los bloques de un archivo pueden estar dispersos por todo el equipo de almacenamiento, los archivos podrán crecer mientras existan bloques libres.
2. Elimina el problema de la fragmentación externa.

Desventajas:

1. No permite el acceso directo; para acceder al bloque n hay que recorrer los n-1 bloques que le preceden.
2. La dispersión de los bloques hace que el acceso sea más lento en general.
3. Gasta espacio adicional debido a que cada bloque contiene un campo dedicado al puntero, donde no se guardan datos útiles desde el punto de vista del usuario.

Este tipo de asignación, al tratar de resolver los problemas de la anterior, empeora la situación dado que un sistema de archivo que solo permita acceso secuencial a sus datos se hace extremadamente lento e ineficiente.

Asignación indexada

La idea básica se aprecia en la figura II.9. Obsérvese que cada archivo usa uno de los bloques del volumen (el 4 en el ejemplo) para almacenar las localizaciones de los bloques que pertenecen al archivo. Ese bloque, que contiene las direcciones de los demás bloques del archivo, es el **bloque de índices** del archivo y desde la tabla de directorio se apunta a él. En este caso (al igual que en la asignación enlazada), no es necesario conocer la longitud del archivo para establecer dónde termina (aunque sí es bueno que esa información esté disponible para otros fines).

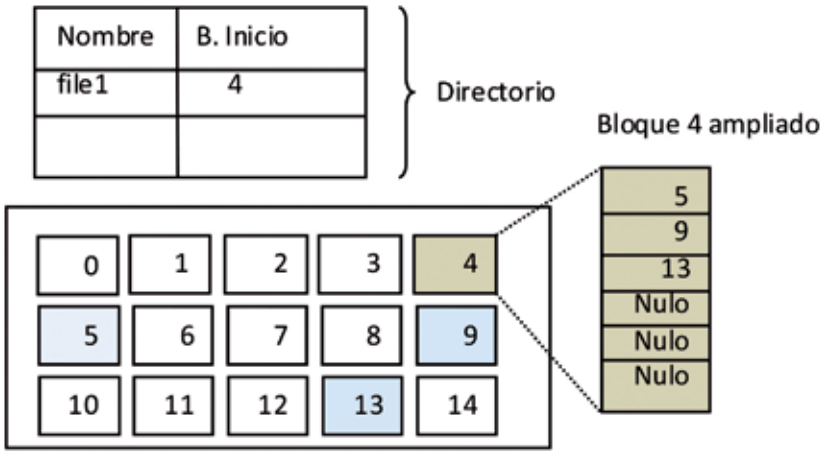


Figura II.9. Asignación indexada. Fuente elaboración propia.

En el caso de la figura II.9, el bloque 4, que es el bloque de índices, no contiene datos sino punteros a otros bloques. Por ese motivo, se conoce que el archivo file1 está compuesto por los bloques 5, 9 y 13. Debe observarse que el bloque 4 lo usa el SO para localizar los bloques del archivo y, por tanto, no contiene ningún dato del archivo en sí.

La solución que ofrece el SO Unix es prácticamente así, mientras que la que ofrece la familia Windows, con su sistema de archivo FAT[29], difiere un tanto dado que existe una gran tabla de índices para todos los archivos del sistema de archivo de un volumen dado. De hecho, dentro de la FAT existe una lista enlazada que es necesario recorrer para localizar los bloques de datos, de ahí que algunos autores la clasifiquen como asignación enlazada cuando en realidad no lo es.

La asignación indexada es la más utilizada actualmente, debido a que resuelve los problemas críticos. Entre sus ventajas se pueden enumerar las siguientes:

1. No provoca fragmentación externa.
2. Permite el acceso aleatorio.
3. Los archivos pueden crecer mientras haya espacio y forma de hacer referencia a sus bloques.

Las malas noticias son que el contenido de un archivo, por lo general, está disperso, lo cual hace que los accesos sean más lentos que en la asignación contigua, pero las desventajas de esta última son demasiado serias como para no tomarlas en cuenta.

II.7 CONTROL DEL ESPACIO LIBRE

Para controlar el espacio libre, se pueden usar varios tipos de estructuras de datos. El sistema de archivo FAT usa la misma estructura para referirse a los espacios libres y

29 Tabla de asignación de archivos (File Allocation Table).

a los bloques de los archivos, pero en realidad, y pese a la popularidad de los SO de la familia Windows, esa forma de llevar el control de espacios libres no es la más eficiente.

Control de espacio libre a través de una tabla

El espacio libre se puede controlar con una **tabla de espacios libres** que tenga dos campos: el primero contiene la dirección del primer espacio libre y el segundo contiene su tamaño. Observe la figura II.10.

DIRECCIÓN DE INICIO	TAMAÑO
10	23
80	12

Figura II.10. Tabla de espacios libres. Fuente elaboración propia.

Cuando se necesita un espacio, el sistema de archivo consulta esa tabla y toma el que se necesita; después, debe actualizar la tabla eliminando la entrada que tomó (si es que se ocupó totalmente) o restableciendo el inicio y el tamaño si solo se tomó parte de la entrada. Este esquema se usa en algunos sistemas de asignación (colocación) contigua y tiene alguna sobrecarga, dado que cada vez que se libere un espacio no basta con ponerlo en la tabla de espacios libres, sino que habrá que verificar si existía algún espacio antes y después de él, los cuales deben eliminarse de la tabla para "unirlos" en un solo hueco.

Control del espacio libre usando un mapa de bits

Una mejor solución es tener un **mapa de bits** (figura II.11), que tendrá una longitud igual a la cantidad de bloques que tenga el volumen, de modo que a cada bit del mapa corresponde a uno y solo un bloque del volumen. Se establece un convenio, la mayoría de las veces, un 0 en la posición n del mapa de bits significa que el bloque n está libre y un 1 significa que está ocupado.

Para satisfacer una solicitud de n bloques, habrá que buscar n bits del mapa de bit con valor 0.

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
1	1	1	0	1	0	0	0	1	1	0	0	0	0	0	1	1	1	1	0

Figura II.11. Mapa de bits para control de espacios libres. Fuente elaboración propia.

En la figura II.11, los bloques 3, 5, 6, 7, 10, 11, 12, 13, 14 y 19 están libres y los restantes están ocupados. En caso de que se fuera a satisfacer una petición que necesitara 10 bloques libres, ocurriría lo siguiente:

* En un sistema de archivo de colocación enlazada o indexada, se podría satisfacer debido a que existe esa cantidad de bloques libres (no importa si están o no contiguos).

* En un sistema de archivo de colocación contigua, esa petición no podría satisfacerse debido a que la máxima cantidad de bloques libres en forma contigua que existe es cinco. Aunque existen cuatro "huecos" libres que sumados dan 10: el primero, que tiene un solo bloque (el 3); el segundo, que tiene tres bloques (5, 6 y 7); el tercero, que tiene cinco bloques (10, 11, 12, 13 y 14); y el último, que tiene un bloque (el 19). En este caso, se aprecia el fenómeno de la fragmentación externa.

Desfragmentar

En los sistemas de archivos contiguos, es necesario desfragmentar cuando el espacio libre queda constituido por "huecos" de diferentes tamaños que no satisfacen las solicitudes. En ese caso, la desfragmentación (para muchos autores es una compactación) se refiere al hecho de agrupar todo el espacio libre en un solo hueco.

En los sistemas enlazados e indexados, no es necesario desfragmentar, debido a que los espacios libres se ven como bloques individuales que pueden usarse de cualquier forma, dado que los archivos pueden estar en bloques dispersos. Cuando los bloques están dispersos, existe la necesidad de realizar múltiples movimientos del cabezal de lectura/escritura (cuando es un disco), lo que resulta en accesos más lentos. Para ayudar a resolver ese problema, la mayoría de los SO que tienen sistema de archivos indexados ofrece programas especiales para desfragmentar los soportes de información. En este caso, la desfragmentación es la acción de hacer que todos los archivos (si es que es posible) estén en forma contigua.

II.8 EJEMPLOS DE SISTEMAS DE ARCHIVOS

En esta sección, se describen varios sistemas de archivos de diferentes SO. El objetivo es analizar cómo se instrumentan las ideas generales que se han discutido hasta el momento.

II.8.1 Sistemas de la compañía Microsoft

Los SO de la compañía Microsoft disponen de dos sistemas de archivos: el FAT y el NTFS.

El sistema de archivo File Allocation Table (FAT)

El sistema de archivo FAT fue el primero usado por la compañía Microsoft y lo instrumentó sobre el SO MS-DOS (Microsoft Disk Operating System). El sistema aún se usa, sobre todo en soportes como las memorias *flash*. Aunque se puede usar en discos duros, debe tomarse en cuenta que no es un sistema seguro.

La figura II.12 muestra las dos estructuras de datos que soportan el sistema de archivo FAT. En la parte superior, se observa la tabla de directorio, mientras que la parte inferior muestra la estructura de datos que da nombre al sistema y que se conoce como FAT.

NOMBRE	EXTENSIÓN	ATRIBUTOS	RESERVADO	HORA	FECHA	CLÚSTER INICIAL	LONGITUD
SO. doc						12	
Prueba						23	

	0	1	2	3	4	5	6	7	8	9
0	0	0	eof	8	0	0	0	0	2	0
1	0	0	3	0	0	0	0	0	0	0
2	0	0	0	30	0	0	0	0	0	0
3	31	38	0	0	0	0	0	0	50	0
4	0	0	0	0	0	0	0	0	0	0
5	eof	0	0	0	0	0	0	0	0	0
6	0	0	0	0	0	0	0	0	0	0
7	0	0	0	0	0	0	0	0	0	0
8	0	0	0	0	0	0	0	0	0	0
9	0	0	0	0	0	0	0	0	0	0

Figura II.12. Sistema de archivo FAT original. Fuente elaboración propia.

La tabla de directorio de este sistema de archivos brinda los siguientes datos acerca de los archivos: su **nombre**, la **extensión** (forma parte de la identificación del archivo y se asocia a su contenido), los **atributos** (oculto, de lectura solamente, sistema y archivo), la **hora y fecha** de actualización, su **longitud** y un campo muy especial para especificar cuál es el **bloque**[30] **inicial** del archivo, es decir, el primer bloque del archivo.

La tabla FAT se localiza muy cerca del inicio del volumen y en un lugar fijo (son dos copias por si se daña una). El directorio raíz también se localiza en un lugar fijo, todo lo cual se establece cuando se formatea el volumen.

Obsérvese que la tabla de directorio que usó inicialmente este sistema de archivo también dejó un campo reservado para futuras ampliaciones. Cuando el sistema FAT quiso extender los nombres de los archivos más allá de la convención 8+3 (ocho caracteres para el nombre y tres para la extensión), la práctica demostró que el espacio reservado era muy pequeño.

La estructura de datos FAT tiene un doble propósito: el primero es conocer la localización de los bloques de datos de cada archivo y el segundo es controlar el espacio libre.

En el ejemplo presentado, la tabla de directorio indica que el bloque inicial del archivo SO.doc es el 12, esa especie de puntero le sirve al SO para buscar la posición 12 en la FAT. La posición 12 de la FAT contiene un número que indica el próximo bloque de ese

30 Clúster según la terminología Windows.

archivo (es el 3); y a su vez, la entrada 3 de la FAT sirve como nuevo puntero y señala al clúster 8 como el próximo, el cual señala al 2 y en la posición 2 de la FAT se ha situado un número especial que marca el fin del archivo (representado simbólicamente por eof-*end of file*). Es decir, los bloques sobre los que reside el archivo SO.doc son: 12, 3, 8 y 2 (en ese orden). De igual forma, el archivo prueba comienza en el bloque 23 y termina en el 50.

La FAT original dejó los dos primeros clústeres del disco para uso del SO (observe de nuevo la figura II.12)

Ejercicio:

* Encuentre la secuencia de bloques que forman parte del archivo prueba.

Las posiciones de la tabla FAT que tienen un cero especifican que los bloques referidos están libres. Debe quedar claro que el sistema FAT no es más que una tabla de apuntadores a bloques y el SO tendrá que hacer los cálculos para saber en qué pista y sector del disco (por ejemplo) está físicamente el bloque n; para ello, usa información que está al inicio del disco (sector del boot) y que especifica, entre otras cosas, el tamaño de los bloques.

Sistema de archivo New Technology File System (NTFS)

El sistema NFTS también organiza los volúmenes en archivos y directorios, pero en esta forma organizativa no existe ningún objeto especial en el disco, ni ninguna localización específica (como el FAT). Además, no existe una dependencia en relación con el *hardware* como es la longitud de 512 bytes de los sectores.

Debe observarse que la existencia de algún objeto especial (como la FAT) hace que el daño en dicho objeto sea catastrófico para el sistema de archivo.

Los diseñadores del sistema de archivo NTFS se trazaron la meta de hacer un sistema confiable que soportara los requisitos del estándar POSIX y que dejara atrás las limitaciones del sistema FAT.

NTFS es un sistema de archivos recuperable debido a que mantiene un registro de las transacciones hechas en el sistema de archivos que se conoce como registro por (o de) diario (*journaling*).

El registro de diario es una bitácora que permite almacenar información para restablecer los datos afectados por las transacciones. Durante el tiempo que demora una transacción, se hace lo siguiente:

1. Se bloquean las estructuras de datos afectadas por la transacción. De esta forma, ningún proceso distinto al que hace la transacción puede modificar las estructuras de datos involucradas en la operación.

2. Se reserva un recurso para almacenar el diario (*journal*), por ejemplo, algunos bloques de disco. En esos bloques, se guarda el estado del volumen antes de la operación, de modo que si ocurre un fallo (de energía, del SO, etc.), se pueda regresar al pasado.

3. Se efectúan las modificaciones en las estructuras de datos (una a una) y para cada modificación:

a. Se apunta en el diario cómo deshacerla, después de lo cual se realiza la modificación.

b. Si se cancela la transacción, se deshacen los cambios, uno a uno, y se borra la traza guardada en el diario.

4. Si se logra realizar la transacción, se borra el diario y se desbloquean las estructuras de datos.

La longitud de los archivos y volúmenes NTFS puede ser de hasta 2^{64} bytes (16 exabytes)[31]. No es recomendable usar NTFS en volúmenes menores que 400 MB, debido a que la sobrecarga que agrega el sistema no se justifica para un volumen de esa dimensión.

En los volúmenes formateados con el sistema de archivo NTFS, se crean varios **archivos de sistema** que pueden alojarse en cualquier parte del volumen con el objetivo de que no suceda como en el sistema FAT, donde un daño en el área que ocupa la FAT hace que el sistema de archivo sea inaccesible.

Los archivos de sistema o *metafiles* son (entre otros):

* $attrdef. Tabla de definición de atributos (*attributes definition*). Contiene todos los atributos del volumen (pueden ser del sistema o definidos por usuarios).

* $badclus. Clúster dañados (*bad clusters*). Contiene la relación de los clúster dañados del volumen, los cuales no pueden usarse.

* $bitmap. Mapa de bits de clúster asignados (*cluster allocation bitmap*). Contiene un mapa de bits del volumen, que especifica los clúster asignados y libres. En el sentido de mantener la relación de clúster asignados y libres, es el equivalente del FAT, pero no se usa para localizar los archivos.

* $boot. Archivo boot (*boot file*). Si el volumen es cargable ("bootable"), contiene el programa de arranque (*bootstrap*).

* $logfile. Archivo de registro (*log file*). Se usa con el propósito de recuperación.

* $mft. Tabla maestra de archivos (*master file table* o MFT). Contiene un registro para cada uno de los archivos en el volumen (archivo, directorios y ***metafile***). Los datos de cada registro incluyen metadatos, tales como: nombre, fecha de creación, permisos de acceso (se usa una lista de control de acceso para esto último) y longitud.

  Si el archivo referido es un directorio, la entrada contendrá el nombre y un identificador de archivo, que es el número de registro que representa el archivo en la tabla maestra de archivos.

  La tabla MFT soporta algoritmos para minimizar la fragmentación del disco.

* $mftmirr. Tabla maestra de archivos 2 (*master file table* 2 o MFT2). Es un espejo de la MFT.

31 Un **exabyte** (EB) equivale a 10^{18} bytes, es decir, 10^6 TB = 10^9 GB.

* $quota. Tabla de cuotas (*quota table*). Indica la cuota de espacio en disco para cada usuario.
* $upcase. Tabla de mayúsculas (*uppercase table*). Se usa para convertir los caracteres en mayúscula, y en minúscula al conjunto de caracteres en mayúscula del código Unicode.
* $volume. Volumen (*volume*). Contiene información del volumen (nombre, versión, etc.).
* $Secure. Archivo de seguridad (*secure file*). Contiene un descriptor único de seguridad para todos los archivos del volumen.
* Además, la referencia punto (.) que es un índice al directorio raíz del volumen.

Archivos y torrentes (*streams*):

Toda la información acerca de los archivos se almacena en registros MFT: su nombre, longitud, posición en el disco, etc., para lo cual se pueden usar uno o varios registros MFT que no tienen que estar contiguos.

Los directorios

Un directorio NTFS es un archivo que especifica referencias a otros archivos y directorios. Está dividido en bloques y cada uno de ellos contiene un nombre de archivo, el atributo base y la referencia al elemento MFT que contiene la información completa del directorio. La estructura interna del directorio es un árbol binario, lo cual permite búsquedas más rápidas.

II.8.2 Sistema de la familia Unix

Los SO de la familia Unix pueden usar diferentes sistemas de archivos, entre los que se destacan el sistema de archivo UFS y el extX.

Generalidades de los sistemas de archivos Unix

Un archivo Unix es simplemente una secuencia de bytes. Las dos estructuras de datos para localizar un archivo son:

* La tabla de directorio.
* El nodo-i (*i-node*).

La figura II.13 da una idea general de esta organización. En este caso, se muestra un disco con una tabla de directorio que contiene dos archivos ordinarios (f1 y f2) y un directorio (D1). Cada entrada de la tabla de directorio apunta a un nodo-i que se detallará más adelante, por ahora es importante comprender que existe un nodo-i para cada archivo o directorio.

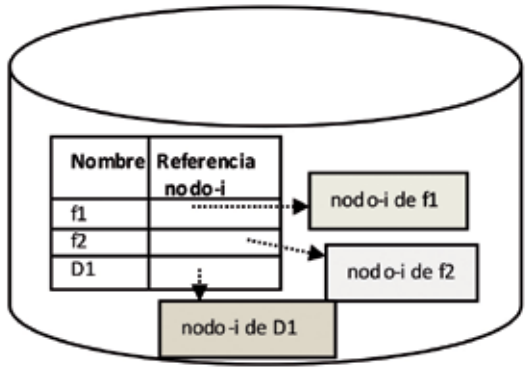


Figura II.13. Relación entre la tabla de directorio y el nodo-i. Fuente elaboración propia.

La tabla de directorio es una estructura de datos que solo contiene dos campos:

* El primer campo contiene el nombre del archivo.
* El segundo campo contiene un puntero, o referencia, a otra estructura de datos, denominada **nodo-i**.

Aunque D1 es un subdirectorio del directorio que se muestra como ejemplo, no existe mucha diferencia entre él y los archivos f1 y f2, debido a que un subdirectorio no es más que un **archivo especial** que contiene referencias a otros archivos (su contenido es en realidad una tabla de directorio).

Algunos nombres de directorios son estándar en todas las versiones de UNIX (al menos en la mayoría), por ejemplo:

* /bin. Contiene los comandos más comunes.
* /dev. Contiene manipuladores de dispositivos (*device drivers*) especiales que controlan el acceso a los periféricos.
* /etc. Contiene programas y archivos de datos del sistema. Los archivos en el directorio /etc/default contienen información que el sistema usa por defecto.
* /lib. Contiene bibliotecas para el lenguaje C y otros lenguajes de programación.
* /mnt. Es un directorio vacío reservado para montar sistemas de archivos.
* /tmp. Contiene archivos temporales creados por programas del SO. Los archivos están presentes cuando se está ejecutando el programa.
* /usr. Aloja los directorios de todos los usuarios del sistema y otros directorios que contienen comandos y archivos de datos adicionales.

En muchos SO, los directorios contienen una gran cantidad de información acerca de los atributos de los archivos. Todo SO necesita de esa información, pero para el caso de los sistemas de archivo que se están analizando, el directorio solo contiene dos

campos (figura II.13); el resto de la información acerca de un archivo dado está dentro del nodo-i.

El nodo-i (*i-node*) o bloque de índices

El nodo-i es una estructura muy importante dentro del sistema de archivo. Cada archivo posee un nodo-i que se referencia desde la tabla de directorio. Esa estructura de datos posee mucha información acerca del archivo y, además, contiene un conjunto de punteros que permiten encontrar los lugares del disco donde está almacenado el archivo.

La figura II.14 muestra una idea esquemática del nodo-i; los primeros campos contienen información acerca del archivo, por ejemplo, su tamaño y la fecha de creación, entre otras cosas importantes.

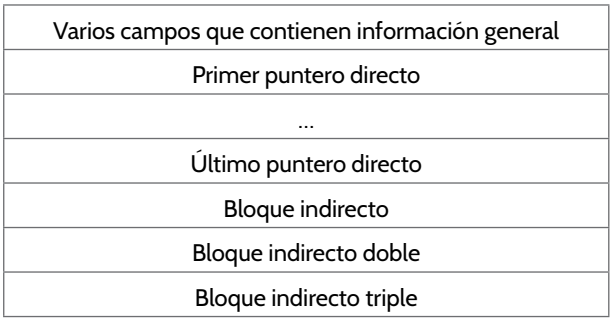


Figura II.14. El nodo-i. Fuente elaboración propia.

Los punteros a bloques del archivo están representados en la figura II.14 por los campos:

* "Primer puntero directo,…, Último puntero directo". Esos campos (usualmente 12, pero pueden variar entre diferentes implementaciones de Unix) apuntan directamente a bloques de datos desde el mismo nodo-i.
* El "Bloque indirecto" se usa cuando un archivo posee más bloques que los que se pueden apuntar directamente desde el nodo-i. En ese caso, este campo apunta a un bloque que solo contiene índices, los cuales apuntan a bloques de datos.
* El "Bloque indirecto doble" se usa cuando no son suficientes los dos direccionamientos anteriores. El campo contiene un puntero que apunta a un bloque de índices, que a su vez apunta a otros bloques de índices desde donde se apunta a los datos, produciendo un doble direccionamiento.
* El "Bloque indirecto triple" indica una tercera indirección.

La figura II.15 da una idea gráfica de la forma en que se usan los punteros y el nodo-i.

Un archivo puede estar referenciado desde varios directorios, lo único que necesita el SO es que desde los distintos directorios se apunte al mismo nodo-i. Por ese motivo, también existe un campo dentro del nodo-i que es un contador y controla la **cantidad de referencias** que hay hacia un archivo determinado. Cuando se recibe la orden de

borrar un archivo, lo que se hace es restarle uno a ese contador en el nodo-i y quitar la entrada en el directorio correspondiente. El archivo se borrará físicamente cuando el contador sea cero.

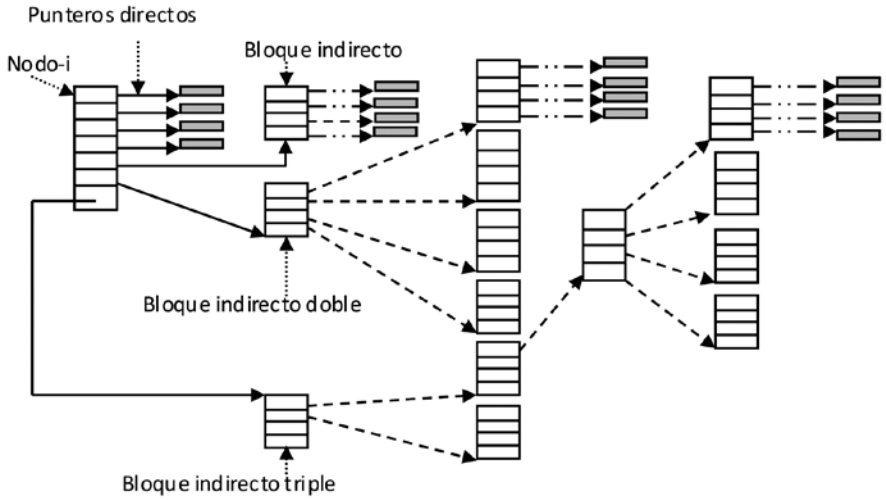


Figura II.15. Los campos apuntadores del nodo-i. Los bloques con color son de datos. Fuente elaboración propia.

El SO tiene, al menos, un sistema de archivo que se denomina *root* y se representa por una diagonal (/). Ese sistema de archivo se crea durante el proceso de instalación y contiene todos los programas y directorios del SO, además de otros directorios de usuarios.

A continuación, se presentan dos comandos que se pueden usar en el SO y que ayudan a comprender algunos conceptos (por convenio, lo que se presenta en negrita lo escribe el SO).

Ejemplos de uso de los comandos df y du

El comando df muestra la utilización del equipo de almacenamiento masivo. Si no se le proporciona un nombre, la información se refiere a todos los sistemas de archivos que estén montados[32] en ese momento. En el ejemplo que sigue, se usa el comando df con las opciones: -P (*paragraph*) para que brinde la información con un cierto formato y -h (*human*) para que la información sea más fácil de entender.

```
mlezcano@hercules:~$ df -P -h
Filesystem      Size  Used Avail Use% Mounted on
/dev/sda1       927M  153M  727M  18% /
tmpfs           503M     0  503M   0% /dev/shm
/dev/sdb5       112G   25G   88G  22% /home
/dev/sda7       927M  8.3M  871M   1% /tmp
/dev/sda5       9.1G  2.1G  6.5G  25% /usr
dev/sda8        60G   1.1G   56G   2% /var
mlezcano@hercules:~$
```

32 Integrar el sistema de archivos de un dispositivo en el árbol de directorio.

Como el comando no especifica ningún nombre, la información se refiere a los sistemas de archivos montados en el equipo. La primera columna contiene sus nombres; la segunda informa sus tamaños; la tercera y la cuarta, los espacios utilizados y los disponibles (respectivamente); la quinta, el porcentaje de uso del sistema de archivo; y la última, el lugar donde está montado cada sistema de archivo, por ejemplo, /dev/sda1 está montado en el directorio raíz (/).

```
mlezcano@hercules:~$ df -i
Filesystem      Inodes   IUsed   IFree     IUse%  Mounted on
/dev/sda1       245760   14554   231206    6%     /
tmpfs           128525   1       128524    1%     /dev/shm
/dev/sdb5       0        0       0         -      /home
/dev/sda7       245760   502     245258    1%     /tmp
/dev/sda5       2443200  113653  2329547   5%     /usr
/dev/sda8       16121856 10325   16111531  1%     /var
mlezcano@hercules:~$
```

En el ejemplo anterior, se usa el comando df para conocer información acerca de los nodos-i de cada sistema de archivo, para lo cual se utiliza la opción -i. Las columnas contienen los nombres de los sistemas de archivos y la cantidad de nodos-i (Inodes) de cada uno de ellos, especificando: los nodos-i usados (IUsed), los libres (IFree) y el porcentaje de uso (IUse%); por último, se aprecia el lugar donde están montados los sistemas de archivos (Mounted On).

El comando du reporta la cantidad de espacio usado por archivos y directorios; si no se especifica un nombre, el reporte se refiere al directorio actual. Por defecto, el espacio se expresa en unidades de 1024 bytes.

En el siguiente ejemplo, se usa el comando du para obtener un reporte del directorio actual.

```
mlezcano@hercules:~$ du
4       ./.mc/cedit
20      ./.mc
324     ./Bash/Lab4
324     ./Bash
16      ./Perl/C6
24      ./Perl/C7
20      ./Perl/C8
60      ./Perl
548     .
mlezcano@hercules:~$
```

El sistema de archivo UFS (UNIX File System)

El sistema UFS es un descendiente del sistema de archivo original que usó Unix Versión 7 y también se conoce por otras denominaciones: Berkeley Fast File System, BSD Fast File System y FFS.

Un volumen UFS está compuesto por las siguientes partes:

* Una cierta cantidad de bloques al comienzo de la partición que se reservan para bloques de inicialización (*boot block*).
* Un **superbloque** que contiene un número mágico que identifica el volumen como un sistema de archivo UFS, así como otros números que describen la geometría del sistema de archivo, las estadísticas y los parámetros de afinación de comportamiento.
* Una colección de grupos de cilindros. Cada grupo de cilindros tiene los componentes siguientes:
  - Una copia de resguardo (*backup*) del superbloque.
  - Un encabezado con estadísticas, listas de bloques libres, etc., acerca del cilindro (similar a las que contiene el superbloque).
  - Una cantidad de bloques para nodos-i y datos. Los bloques para nodos-i se numeran secuencialmente, los primeros se reservan (por razones históricas) y están seguidos de un nodo-i especial para el directorio raíz.

Las primeras versiones del sistema de archivo Unix se denominaban simplemente FS (File System) e incluían el *boot block*, el superbloque, una cierta cantidad de nodos-i y bloques de datos. Este formato era adecuado para los pequeños discos que manejaban los SO Unix de aquella época, pero con los avances de la tecnología los discos crecieron y el formato original comenzó a provocar movimientos excesivos de los cabezales de lectura-escritura (*thrashing*), lo que llevó a la concepción de los grupos de cilindros (con sus nodos-i y bloques de datos) para reducir este inconveniente[33].

Actualmente, existen diferentes variantes de UFS implementadas por distintos propietarios de sistemas Unix, tales como: SunOS/Solaris, System V Release 4, HP-UX, Tru64 UNIX, 4.4BSD, BSD Unix systems FreeBSD, NetBSD, OpenBSD y DragonFlyBSD.

El sistema de archivo extendido "ext"

El SO Linux incluye una implementación de UFS para lograr compatibilidad al nivel de lectura binaria con otros Unix, pero no es una implementación estándar.

Ext (*extended file system*)[34] fue el primer sistema que se creó específicamente para los SO Linux y se inspiró en UFS. Después de un tiempo, ext fue reemplazado por otros dos sistemas: ext2 y xiaf.

El sistema de archivo extendido "ext2"

Es un sistema de archivos para el *kernel* de Linux. Su principal desventaja es que no implementa el registro por diario (*journaling*) que sí implementa su sucesor ext3.

33 Ideado por Marshall Kirk McKusic para el 4.2BSD'S FFS (Fast File System).

34 Fue diseñado por Remy Card para resolver limitaciones de Minix.

Ext2 (*second extended file system*) fue el sistema de archivo por defecto de las distribuciones de Linux Red Hat, Fedora Core y Debian, hasta que fue reemplazado por su sucesor ext3.

El sistema de archivo tiene una tabla de tamaño fijo, donde se almacenan los nodos-i. El tamaño de los bloques se puede especificar cuando se crea el sistema de archivos (desde 512 bytes hasta 4 KB).

El sistema de archivo extendido "ext3"

El sistema de archivo ext3 (*third extended file system*) soporta el registro de archivos por diario (*journaling*) y se usó extensivamente en las distribuciones Linux. Hoy en día, está siendo sustituido por su sucesor ext4.

La principal diferencia con ext2 es, precisamente, el registro por diario. Los sistemas ext3 utilizan un árbol binario balanceado (AVL)[35] e incorporan el sistema Orlov[36] para asignar los bloques de disco. En los árboles AVL, la altura de la rama izquierda no difiere en más de una unidad de la altura de la rama derecha o viceversa, lo cual permite agilizar la búsqueda.

La velocidad que se logra en los sistemas ext3 es menor que en los sistemas JFS[37], ReiserFS[38] y XFS; el sistema también es menos escalable que estos últimos, pero se puede actualizar de un sistema de archivos ext2 a uno ext3 sin perder datos, lo cual es una innegable ventaja en relación con los otros sistemas de archivos mencionados en este párrafo.

El sistema de archivos ext3 permite direccionar hasta 2^{32} bloques, que pueden tener diferentes tamaños.

El sistema de archivo extendido "ext4"

El sistema de archivo ext4 (*fourth extended file system*), también permite el registro por diario o bitácora (*journaling*). Su primera aparición fue de orden experimental; desde el 2008, dejó de ser un experimento que se convirtió en el sustituto de ext3 y es compatible con este último.

El sistema de archivos ext4 es capaz de trabajar con volúmenes de hasta un exbibyte y con archivos de hasta 16 TiB.

Introduce el concepto de ***extents***, que es distinto al esquema de bloques. Un *extent* es un conjunto de bloques físicos contiguos que se pueden asignar como un todo, lo que mejora el rendimiento cuando se trabaja con archivos grandes al reducir la fragmentación.

En ext4, es posible hacer pre reserva de espacio en disco, para lo cual se usa la llamada al sistema preallocate(). El espacio pre reservado para un archivo tiene una buena probabilidad de quedar contiguo.

Otra de las ventajas de ext4 es la asignación retrasada de espacio en disco. Esta técnica retrasa la reservación de bloques del disco hasta que esté cerca el momento de

35 El nombre del algoritmo AVL se deriva de los apellidos de sus inventores: Georgii Adelson-Velskii y Yevgeniy Landis.

36 El sistema para asignar bloques Orlov se originó en BSD.

37 JFS (Journaling File System). Sistema de archivos de 64-bit con registro de diario.

38 ReiserFS toma el nombre de su autor (Hans Reiser). Sistema de archivos de propósito general con *journaling*.

escribirla (en otros sistemas de archivos la reserva se hace antes), con lo cual se mejora el rendimiento y se reduce la fragmentación.

En este sistema de archivo, el nivel de profundidad de los subdirectorios puede ser 64 000 (en ext3 era 32 000).

Existen muchas otras ventajas de este sistema de archivo que deberán ser consultadas en bibliografías específicas.

II.9 RESUMEN DEL CAPÍTULO

El sistema de archivo administra los volúmenes de almacenamiento.

La arquitectura del sistema de archivo está estructurada por niveles, desde el más cercano a los equipos en sí, o de bajo nivel, hasta el más cercano a los usuarios; en este último nivel, el sistema hace una abstracción de las características físicas de los equipos para presentar los datos agrupados bajo el concepto de archivo.

Los archivos se organizan en directorios, lo que permite localizarlos más fácilmente, a la vez que proporciona una visión más organizada de los volúmenes.

El sistema de archivo debe tener una forma efectiva de controlar los espacios libres y ocupados del volumen, y en general, se destacan tres técnicas para asignar espacios: contigua, enlazada e indexada.

Los sistemas de archivos varían de un SO a otro; cabe destacar los sistemas NTFS y FAT de los SO de la familia Windows y los sistemas UFS y extX de la familia Unix.

La asignación de espacio en disco en los sistemas indexados (son los más usados hoy en día) provoca fragmentación y por eso muchos sistemas de archivos incluyen utilitarios para desfragmentar los discos, con lo cual se mejora el rendimiento del sistema de archivo.

II.10 EJERCICIOS

1. El SO Windows implementa dos tipos de sistemas de archivos: FAT y NTFS. El sistema FAT fue el primero que se implementó y aún hoy se usa, aunque no es un sistema de archivo seguro.

   a. Haga un algoritmo para borrar archivos en el sistema FAT.

   b. El comando chkdsk permite, entre otras cosas, encontrar los clúster perdidos de un equipo de almacenamiento externo. Los clúster perdidos son aquellos que están referenciados como usados en la FAT (tienen un número diferente de cero), pero no pertenecen a ningún archivo. Haga un algoritmo que permita encontrar los clústeres perdidos y cree archivos con las cadenas de referencias que encuentre.

2. Los sistemas de archivos pueden ser de colocación contigua, enlazada e indexada.

   a. Defina las estructuras de datos imprescindibles para un sistema de archivos de colocación contigua. Use ideas propias, no debe ser algún sistema real.

   i. Haga un algoritmo para copiar archivos en ese SO.

b. Defina las estructuras de datos imprescindibles para un sistema de archivos de colocación enlazada. Use ideas propias, no debe ser algún sistema real.

   i. Haga un algoritmo para mover archivos en ese SO.

   Nota: debe observar que el algoritmo es menos complejo cuando el movimiento es dentro del mismo disco.

c. Defina las estructuras de datos imprescindibles para un sistema de archivos de colocación indexada. Use ideas propias, no debe ser algún sistema real.

   i. Haga un algoritmo para borrar archivos en ese SO.

3. Haga un algoritmo para borrar archivos en el sistema de archivo UFS del SO Unix.

4. Busque información adicional acerca de los sistemas de archivos: NTFS y ext4.

   a. Haga un reporte acerca de las ventajas y desventajas de cada uno de esos sistemas de archivos.

   b. Haga una valoración crítica de ambos sistemas de archivos.

5. Busque los comandos del SO Unix relacionados con el sistema de archivo.

   a. Haga una tabla, con el formato mostrado a continuación, que le permita buscar ayuda cuando necesite usar alguno de esos comandos.

COMANDOS UNIX		
Comando	Propósito	Ejemplos

6. Busque los comandos del SO Windows 10 relacionados con el sistema de archivo.

   a. Haga una tabla, con el formato mostrado a continuación, que le permita buscar ayuda cuando necesite usar alguno de esos comandos.

COMANDOS WINDOWS 10		
Comando	Propósito	Ejemplos

7. Explique la idea que debe seguir un mecanismo de desfragmentación de disco.

II.11 BIBLIOGRAFÍA CONSULTADA

Abraham, S., Baer- Galvin, P., & Gagne, G. (2013). *Operating system concepts* (9.ª ed.). Nueva Jersey: Jhon Wiley & Sons.

Dhamdhere, D. M. (2008). *Operating systems. A concept based aproach.* Nueva York: McGraw-Hill Education.

Elmasri, R., Carrick, A., Levine, D. (2009). *Operating systems: A spiral approach.* Nueva York: McGraw-Hill.

Gilly, D. (2003). *Unix in a Nutshell.* En *The Unix CD Bookshelf* (3.ª ed.). Sebastopol: O'Reilly Media.

Harvey, M., Deitel, P., & Choffnes, D. (2003). *Operating systems* (3.ª ed.). Nueva Jersey: Prentice Hall.

Microsoft Corporation. (2015). *Introducing Windows 10 for IT Professionals. Preview Edition.* Alburquerque: Microsoft Press.

Peek, J., O'Reilly, T., & Loukides, M. (2003). *UNIX Power Tools.* En *The Unix CD Bookshelf* (3.ª ed.). Sebastopol: O'Reilly Media.

Peek, J., Todino G., & Strang, J. (2003). *Learning the Unix Operating System.* En *The Unix CD Bookshelf* (3.ª ed.). Sebastopol: O'Reilly Media.

Shotts Jr., W. (2012). *The Linux Command Line: A complete introduction.* San Francisco: No Starch Press.

Stallings, W. (2014). *Operating systems: Internals and design principles* (8.ª ed.). Nueva York: Pearson.

Tanenbaum, A. (2006). *Operating systems: Design and implementation* (3.ª ed.). Nueva Jersey: Prentice Hall.

Tanenbaum, A., & Bos, H. (2014). *Modern operating systems* (4.ª ed.). Nueva York: Pearson.

CAPÍTULO III

Administración de la memoria

RESUMEN

En este capítulo, se exponen las técnicas que se usan para administrar la memoria. Comienza haciendo un bosquejo de los esquemas históricos para administrar la memoria y después pasa a explicar los sistemas paginados y segmentados, muy usados en la actualidad, lo que incluye el manejo de la memoria virtual como extensión de la memoria interna. En relación con la memoria virtual, se enfatiza en la demanda de página y de segmentos, a la vez que se muestran y se detallan estructuras de datos tan importantes como las tablas de páginas y de segmentos, además de las estructuras para controlar la memoria ocupada y libre. En el capítulo, se analizan distintos algoritmos de reemplazamiento, detallando sus aspectos positivos y negativos. El capítulo finaliza con un resumen y una sección de ejercicios propuestos.

Palabras clave: administración de la memoria, páginas, segmentos, tablas de páginas, tablas de segmentos, algoritmos de reemplazamiento

¿Cómo citar este capítulo? / How to cite this chapter?

Lezcano-Brito, M. G. (2017). Administración de la memoria. En *Fundamentos de sistemas operativos. Entornos de trabajo* (pp. 105-127). Bogotá: Ediciones Universidad Cooperativa de Colombia.

CHAPTER III

Memory management

ABSTRACT

This chapter discusses the techniques used to manage memory. It begins by outlining the historical schemes to manage memory and then explains paginated and segmented systems, much used today, including management of virtual memory as an extension of internal memory. In relation to virtual memory, page and segment demand is emphasized, while detailing important data structures such as page and segment tables, as well as structures that control used and free memory. In the chapter, different replacement algorithms are studied, detailing their positive and negative aspects. The chapter ends with a summary and a section of proposed exercises.

Keywords: Memory management, pages, segments, page tables, segment tables, replacement algorithms.

La memoria es una parte fundamental de cualquier equipo de cómputo; en sí es un recurso de *hardware* que debe ser administrado por un módulo del SO. Dependiendo de la cantidad de memoria que se disponga y de la técnica que se use para administrarla, se pueden ejecutar más o menos procesos con mayor o menor velocidad.

La memoria se puede ver como un **arreglo de palabras** o **bytes**, cada byte tiene una dirección asociada que permite referirla y usarla. La figura III.1 muestra una memoria de N + 3 bytes, donde N puede ser cualquier valor; por ejemplo, si la máquina dispone de una memoria de 1024 bytes, N es 1020. De esta forma, se puede acceder a cualquier dirección de esa memoria desde la dirección 0 hasta la 1023.

En el ejemplo, el contenido de la dirección 4 es A. Obsérvese la clara distinción que existe entre una **dirección de memoria** y su **contenido**: la primera se refiere a la posición que ocupa un byte específico dentro del arreglo de bytes que es la memoria en sí, mientras que la segunda hace alusión a algo que se almacena en esa localización.

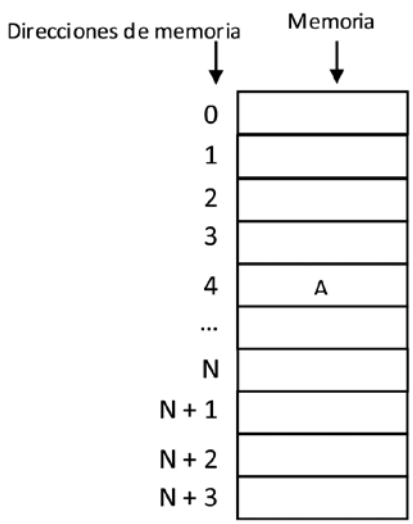


Figura III.1. La memoria vista como un arreglo de bytes. Fuente elaboración propia.

Cuando se esquematiza la memoria, como en la figura III.1, no existe acuerdo, acerca de si la primera dirección (la 0) queda en la parte superior de la figura o en la inferior, en este texto se asumirá el esquema de la figura III.1.

La memoria principal, o básica, y los registros del procesador son los únicos elementos de almacenamiento a los que la CPU tiene acceso, y por eso resulta necesario que la información sobre la que actúa la CPU, esté en alguna de esas partes: en registro o en memoria principal.

Aunque el desarrollo vertiginoso del *hardware* ha potenciado el surgimiento de memorias más veloces y de mayor capacidad, la cantidad de memoria sigue resultando poca para los usuarios de las computadoras (a cualquier nivel), debido a que las aplicaciones y el propio SO se hacen cada vez más voraces al incluir interfaces gráficas más poderosas y otras facilidades que consumen grandes cantidades de memoria. A su vez, las dificultades de escasez de memoria se agudizan cuando se trabaja en un sistema multiprogramado, dado que en ese caso se hace crucial la protección.

El tema central del capítulo está dirigido a estudiar las diferentes técnicas que usa el SO para administrar la memoria, y también se analizan las alternativas para hacerla parecer mayor de lo que es realmente.

III.1 ESQUEMAS HISTÓRICOS DE ADMINISTRACIÓN DE LA MEMORIA

Los primeros esquemas de administración de la memoria ya casi no se utilizan, pero es necesario revisarlos, porque fueron ellos los que introdujeron algunos conceptos importantes que ayudaron a su evolución.

La máquina rasa

El primer esquema que se usó para administrar la memoria era muy simple; consistía en cargar un único programa completo y en forma contigua en la memoria (figura III.2), lo cual tenía dos ventajas dignas de destacar:

1. Le ofrecía mucha flexibilidad a los usuarios, debido a que podían referirse fácilmente a cualquier localización de memoria.
2. Al ser muy simple, se minimizaba el costo de acceder a las distintas localizaciones.

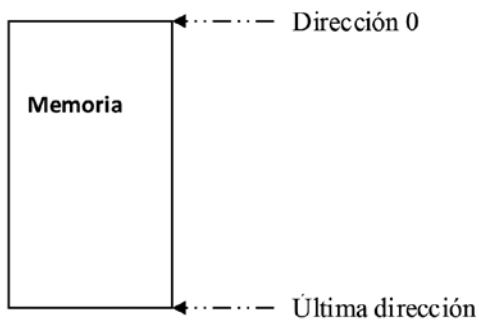


Figura III.2. Máquina rasa. Fuente elaboración propia.

Infortunadamente, esas ventajas estaban lastradas por algunas desventajas aún más importantes, como son las siguientes:

1. No facilitaba la prestación de servicios.

2. El usuario tenía un control completo sobre la computadora.
3. En esa época, no existía nada parecido a un SO y por eso toda la memoria pertenecía a los usuarios, de manera que en ella no existía un componente ajeno a esos usuarios que controlara el uso de la memoria.

En la figura III.2, se puede apreciar que la memoria se ve como algo plano desde la primera dirección hasta la última, pues no hay frontera entre el proceso que se adueñe de la memoria y el SO, y queda claro que el programador tiene acceso a cualquier área de la memoria, lo cual le ofrece una gran flexibilidad para usarla, pero no existe manera alguna para proteger partes del código que no deben cambiarse.

Monitor residente

El esquema del monitor residente (figura III.3) se regía por el mismo principio de cargar los programas totalmente en memoria y en forma contigua, pero en este caso la memoria se dividía en dos áreas:

* En la primera área, se cargaba una parte del SO denominada **monitor residente**. Como su nombre lo indica, ese código residía permanentemente en memoria y tenía la tarea de monitorear (controlar) lo que se hacía.
* La segunda área se asignaba al usuario. Esta implementación se usó por primera vez en el sistema Fortran Monitoring System.

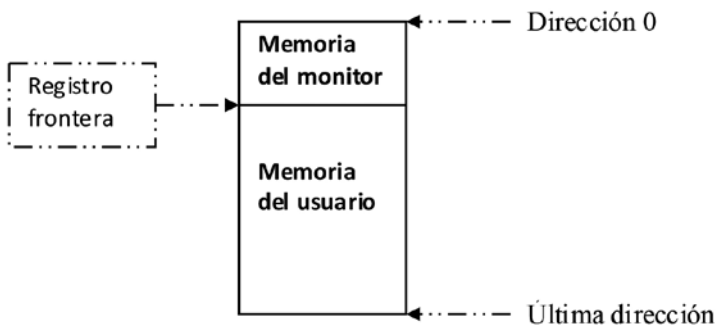


Figura III.3. Monitor residente. Fuente elaboración propia.

Para proteger el monitor residente (su código no debía alterarse), se podía usar una dirección que sirviera de frontera entre la zona en que él residía y la zona del usuario. Cada dirección generada se verificaba para tener certeza de que no se violara esa frontera; si existe una violación, el SO tomaba la decisión adecuada (podía ser tan drástica como abortar el proceso violador).

La dirección de frontera podía cargarse en un registro especial o ser una dirección fija[39], pero esta última idea no resultaba muy atractiva por el hecho de que el SO no podía crecer (ampliaciones futuras podrían necesitar más memoria).

39 Años más tarde, los SO CP/M y MS-DOS, que se implementaron para micro computadoras, usaban direcciones fijas como frontera.

Obsérvese que con esta concepción era necesario **relocalizar** las direcciones de memoria, debido a que las direcciones del proceso que se cargara en memoria no eran **absolutas**, es decir que la dirección 0 de un proceso no comenzaba a partir de la dirección 0 real sino que era **relativa** a la dirección de la frontera.

La atadura o liga de una dirección (el cálculo de su valor real) se podía hacer en varios momentos o tiempos:

* Tiempo de compilación. El compilador debía generar direcciones absolutas cuando traducía el programa del **código fuente** al **código objeto**.
* Tiempo de enlace. El enlazador debía generar direcciones absolutas cuando enlazaba los diferentes módulos que componen una aplicación. Obsérvese que para generar direcciones absolutas en los tiempos de compilación y de enlace era necesario conocer el valor del registro frontera.
* Si no se conocía el valor del registro frontera, la atadura de direcciones se tenía que hacer en tiempo de carga o en tiempo de ejecución. En ambos casos, el compilador debía generar un código que pudiera ser relocalizado (en la jerga de computación se dice que es relocalizable).

En este caso, cambiaba un poco el apoyo del *hardware*, debido a que a cualquier dirección generada por la CPU se le sumaba el valor de la frontera para obtener la dirección física o real a la que hacía referencia la dirección lógica generada. En este esquema de direccionamiento, el usuario nunca trabajaba con direcciones reales.

El **mapeo de direcciones lógicas en direcciones físicas** es el principal trabajo del módulo de administración de memoria de cualquier SO.

Particiones múltiples

Este fue el primer esquema de manejo de memoria para permitir la multiprogramación. La idea era muy básica: la memoria de la computadora se dividía en zonas de varios tamaños denominadas **particiones**; cada proceso debía cargarse en una de las particiones para ejecutarse.

Obsérvese que, al igual que en los esquemas anteriores, debía cargarse el proceso completo y en localizaciones contiguas de memoria.

Cuando un proceso terminaba, su partición quedaba libre y se podía escoger otro de la cola de listos para cargarlo a memoria (si es que el planificador tomaba esa decisión).

El esquema tenía dos implementaciones:

* En la primera, se hacían particiones con tamaño fijo y se denominaba MFT (**M**ultiprogramming with a **F**ixed number of **T**asks).
* En la segunda, las particiones eran variables y se denominaba MVT (**M**ultiprogramming with a **V**ariable number of **T**asks).

Queda claro que en cualquier caso era necesario proteger las fronteras de cada proceso, lo cual se podía hacer si el *hardware* proporcionaba registros para utilizarlos como receptores de los valores fronteras de las particiones. La figura III.4 muestra la idea, en forma de algoritmo, de esta concepción; se supone que el registro frontera inferior (contiene la dirección de menor valor) se denomina RFI y el registro frontera superior (contiene la dirección de mayor valor) se nombra RFS.

```
frontera(direccionRelativa)
  {
    direccionAbsoluta = direccionRelativa + RFI
    Si ((direccionAbsoluta > RFI) && (direccionAbsoluta < RFS))
      Entonces
        acceder a la dirección
      En caso contrario
        error de direccionamiento
  }
```

Figura III.4. Protección de las fronteras. Fuente elaboración propia.

MFT

En el caso de MFT, se pueden tener dos alternativas para la cola de espera: la primera consiste en tener colas separadas por cada una de las particiones, mientras que la segunda es que exista una sola cola para competir por cualquier partición disponible que satisfaga los requerimientos.

En el primer caso, los trabajos se asignan a las colas de acuerdo con su tamaño, tal como se aprecia en la figura III.5. No existe competencia entre las diferentes colas; si la partición de 20 k del ejemplo está libre y existen trabajos en la primera cola, no se asignarán a esa partición.

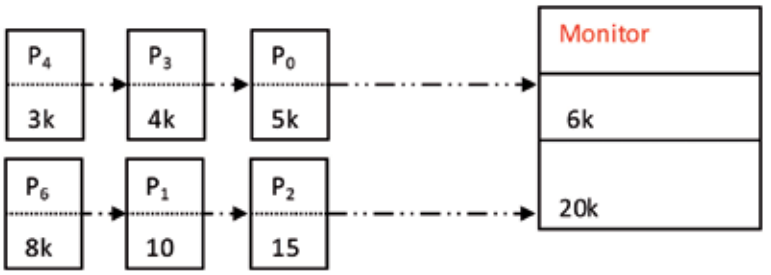


Figura III.5. MFT con colas individuales por particiones. Fuente: Adaptado del libro *Operating System Concepts*.

La segunda variante (figura III.6), aunque permite que los trabajos se asignen a cualquier partición, puede traer por resultado que un trabajo grande esté esperando por uno más pequeño que cabe en una partición menor que está vacía y ocupa la única partición que podría servirle.

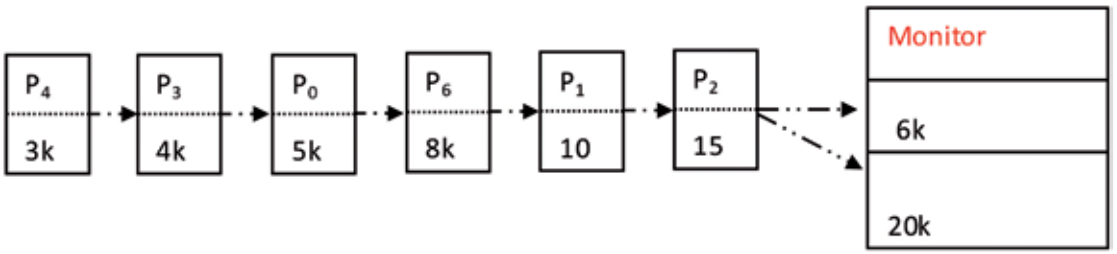


Figura III.6. MFT con una sola cola.. Fuente: Adaptado del libro *Operating System Concepts*.

Queda claro que en esta segunda variante se puede usar la política de "escoger el área disponible mejor", que aunque no resuelve el problema, al menos lo disminuye un tanto.

La filosofía MFT **provoca fragmentación interna** al dejar parte de las particiones sin usar, dado que los requerimientos de los procesos rara vez cubren totalmente la partición a la cual se asignan y el espacio no utilizado no se puede asignar a otro proceso.

MVT

En el caso de MVT, el SO debe mantener una estructura de datos que controle los espacios ocupados y libres de la memoria para poder asignarlos de acuerdo con las peticiones que se hagan. Ante una solicitud dada, se pueden usar tres soluciones (observe la similitud con la forma de colocación contigua de los sistemas de archivos analizada en el capítulo anterior):

* El mejor acceso. Consiste en escoger la partición más pequeña donde quepa el proceso. Esta política tiende a dejar espacios muy pequeños que no permiten la ubicación de nuevos procesos. Se dice que "el mejor acceso" resulta en el peor servicio.
* El peor acceso. Consiste en escoger la partición más grande donde quepa un proceso. En este caso, los espacios dejados como libres son mayores, y por tanto, tienen una mayor probabilidad de volver a ser usados que los dejados por la política del mejor acceso.
* El primer acceso. Consiste en escoger la primera región libre que satisfaga el requerimiento.

El principal problema con MVT es la **fragmentación externa** que se provoca al dejar pequeños espacios libres que, en su conjunto, podrían usarse por algún proceso, pero que al no estar contiguos, no pueden utilizarse. Una solución a este problema es la desfragmentación de la memoria.

La desfragmentación no siempre es posible, solo se puede hacer si el código es relocalizable. También, puede resultar costosa, debido a que para mover los procesos será necesario detenerlos y calcular todas las relocalizaciones. Una consideración a tener en cuenta es mover solo los vecinos del proceso afectado; en todo caso, se trata de mover la menor cantidad de procesos.

III.2 ESQUEMAS ACTUALES DE ADMINISTRACIÓN DE LA MEMORIA

Debido a que han demostrado su eficacia, los sistemas actuales de administración de la memoria datan de varios años, pero con los avances en el desarrollo del *hardware*, muchas funciones, que en un principio se implementaban por *software*, han emigrado hacia el *hardware*.

Como se analizó antes, la estrategia MVT causa fragmentación externa al dejar espacios de memoria libre que no están contiguos, pero que sumados satisfacen las necesidades de un proceso dado. La solución es la desfragmentación, pero eso implica gastos en tiempo de ejecución (a veces considerables). Una mejor solución a esta situación puede ser permitir que los procesos estén en diferentes áreas de memoria no contiguas. Las dos variantes más difundidas que responden a ese esquema son el paginado y la segmentación, aunque también existen combinaciones de estas dos técnicas.

Para implementar el paginado y la segmentación, se hace uso de dos conceptos en relación con la memoria:

* Memoria física. Es la memoria real que posee el equipo de cómputo.
* Memoria lógica. Es la memoria que necesita cada proceso.

III.2.1 Paginado

En esta concepción, toda la memoria física se divide en unidades de igual longitud, *l*, llamadas **marcos de página** (*frames*), mientras que la memoria lógica de los procesos se divide en unidades de la misma longitud, *l*, denominadas **páginas**. Lo anterior permite que las páginas (divisiones de la memoria lógica) quepan en los marcos de páginas (divisiones de la memoria física).

Debe quedar claro que la única memoria que existe es aquella de la que dispone **físicamente** la computadora, en este caso son los marcos de página. De otra parte, las páginas son las unidades en que se divide un programa ejecutable para cargar cada una de esas unidades en los marcos de página disponibles. Normalmente, la cantidad de marcos de página (memoria real o física) es menor que la suma de las páginas que necesitan todos los procesos que se están ejecutando.

Es necesario tener alguna forma de localizar las páginas de los procesos, a fin de saber en qué parte física de la memoria están, para lo cual se usa una estructura de datos llamada **tabla de página**. Cada proceso tiene una tabla de página que especifica cuáles de sus páginas están cargadas y dónde están (en qué marco de página).

Toda dirección generada por la CPU se dividirá de manera que se forme un par (número de página, desplazamiento). El número de página se usa como un índice dentro de la tabla de página y el desplazamiento indica una localización dentro del proceso, tomada a partir de la dirección 0 de la página específica que se esté referenciando.

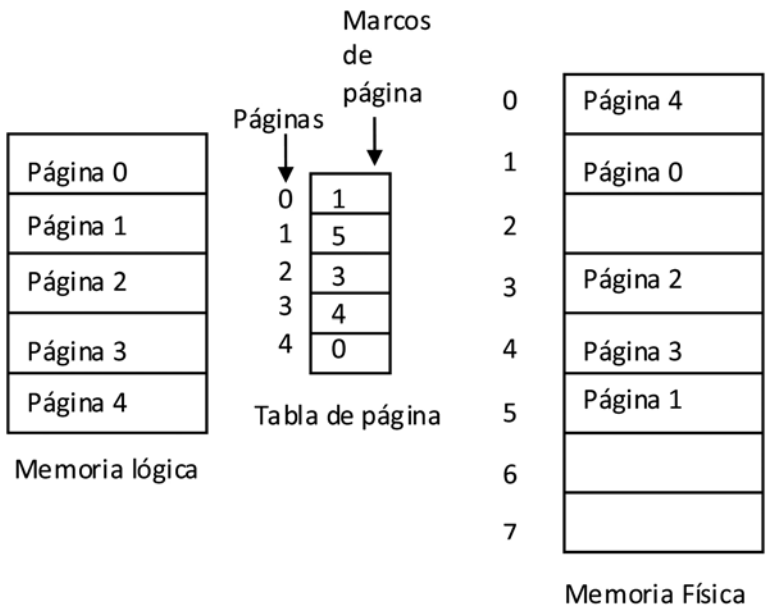


Figura III.7. Esquema paginado.. Fuente: Adaptado del libro *Operating System Concepts.*

En la figura III.7, se puede apreciar un caso particular donde la tabla de página de un proceso dado indica que:

1. La página 0 está en el marco de página 1.
2. La página 1 se ubica en el marco de página 5.
3. La página 2 quedó sobre el marco de página 3.
4. La página 3 está en el marco de página 4.
5. La página 4 ocupa el marco de página 0.

Obsérvese que esta manera de usar la memoria no exige que un proceso resida en localizaciones contiguas, dado que sus páginas pueden estar cargadas en marcos de página que no son vecinos y la tabla de página se usa para calcular las direcciones físicas.

A continuación, se presenta un ejemplo de **traducción** de una dirección lógica a una dirección física. Antes de entrar en detalles, debe señalarse que la longitud de las páginas debe ser potencia de dos.

Para el ejemplo, se asumirá que las páginas son de 2 k, es decir, 2048 bytes y que se desea acceder a la localización 6146 del proceso de la figura III.7; el SO (apoyado por el *hardware* hoy en día) procederá de la siguiente manera:

1. El *hardware* divide la dirección lógica 6146 en forma de un par <página, desplazamiento>, para lo cual hace el siguiente cálculo: 6146 / 2048 = 3 con resto 2,

es decir, esa dirección corresponde a la página 3 desplazamiento 2, y dentro de la memoria lógica del proceso, el par queda en la forma: <3, 2>.

2. Después el sistema busca, en la tabla de página, la localización de la página 3. En este ejemplo, la página 3 reside en el marco de página 4.
3. El paso siguiente es calcular la dirección, en la memoria física de la dirección lógica que se está calculando.
   a. Se calcula la dirección de inicio del marco de página (marco 4 en este caso); como el tamaño de las páginas es 2048, la operación será la siguiente:

$$4 * 2048 = 8192$$

   Lo que significa que la página 3 está cargada a partir de la dirección física 8192.

   b. Por último, se le suma el desplazamiento que tiene esa localización dentro de la página, que en este ejemplo es 2: 8192 + 2 = 8194. La localización física de la dirección lógica 6146 es, en este momento, 8194. Se dice "en este momento" porque puede ser que en un futuro la página 3 sea quitada de la memoria (se verá más adelante) y no ocupe el mismo marco de página cuando se cargue de nuevo.

El sistema debe conocer cuáles partes de la memoria están libres y cuáles están ocupadas; con ese propósito existe una estructura de datos denominada **lista de marcos libres**, que tiene una entrada por cada marco de página en los que se divide la memoria y especifica si un marco dado está libre o está ocupado.

Resulta casi imposible que un proceso ocupe exactamente la memoria que necesita, debido a que el espacio se asigna a nivel de marcos de página; eso hace que, en la mayoría de las ocasiones, la última página de cada proceso solo utilice una parte del marco de página en que reside. El problema anterior provoca el fenómeno de fragmentación interna, que no tiene solución en este esquema de administración de la memoria.

Protección en los esquemas paginados

La protección en un esquema paginado se puede controlar a través de algunos bits que se agregan a la tabla de página, por ejemplo, para especificar si la página es de lectura solamente, de lectura-escritura, etc.

Un bit que tradicionalmente se agrega a las páginas es el **bit de validez**, que especifica si una página dada pertenece o no al conjunto de direcciones lógicas del proceso. Por ejemplo, supóngase que un proceso tiene un espacio de direcciones lógicas que va desde la dirección 0 hasta la 8974 y que se tienen páginas de 2048 bytes, eso significa que el proceso necesita 5 páginas, dado que 8974 / 2048 = 4 y un resto (la última página no necesita un marco de página entero, pero el SO toma el marco entero).

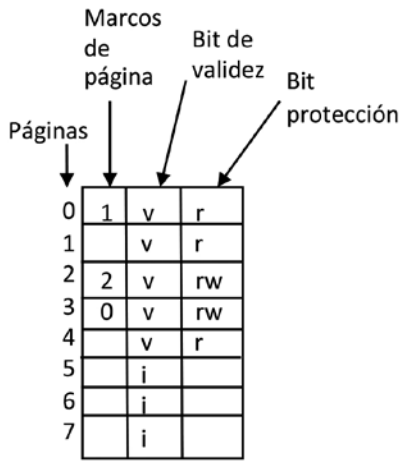


Figura III.8. Tabla de página con bits adicionales.. Fuente: Adaptado del libro *Operating System Concepts*.

Entonces las páginas válidas para ese proceso son las 0, 1, 2, 3, y 4. La tabla de página con el bit de validez más un bit de protección de lectura se aprecia en la figura III.8. Obsérvese que no es necesario que todas las páginas estén cargadas en memoria; en este caso, las páginas 1 y 4 no lo están, ellas (junto a la página 0) son de lectura solamente (r), mientras que las demás son de lectura-escritura (rw).

III.2.2 Segmentación

Para muchos programadores, la forma en que el esquema paginado divide los procesos no resulta nada natural, debido a que ellos tienen la concepción de que un programa está formado como un conjunto de **unidades lógicas**; por ejemplo, cada uno de los procedimientos o las funciones que lo conforman, la pila, un arreglo de datos determinado, etc.

Los sistemas segmentados usan esa idea de los programadores para dividir los procesos en unidades lógicas, las cuales se denominan **segmentos**. Obsérvese que los segmentos, a diferencia de las páginas, tienen distintas longitudes y que en los esquemas segmentados no existe ninguna división, *a priori*, de la memoria física.

Cada unidad lógica, o **segmento**, se asigna o se libera como un todo, lo que va dando lugar a un cierto fraccionamiento de la memoria en partes aisladas de distintas longitudes. Esto puede dar como resultado que una petición de memoria de un tamaño t dado pueda no ser satisfecha a pesar de existir espacios separados, menores que t, que sumados satisfacen la solicitud, lo cual se conoce como **fragmentación externa**.

La fragmentación externa no existe en el paginado debido a que las páginas y los marcos de página son de la misma longitud; de otra parte, la fragmentación interna –que sí existe en el paginado– no existe en el segmentado, debido a que en este último esquema se asigna a cada segmento la cantidad exacta de memoria que necesita.

El espacio de direcciones lógicas es un conjunto de segmentos; cada segmento tiene asociado un **nombre** y una **longitud**, aunque para simplificar la implementación, los segmentos se refieren por un número y no por el nombre, de modo que una dirección lógica la conforma un par <segmento, desplazamiento>.

El compilador, normalmente, construye los segmentos dejando un reflejo del programa compilado; obsérvese que aunque el usuario se refiere a los objetos del programa a través de direcciones compuestas por dos partes, queda claro que la memoria continúa siendo un arreglo unidimensional.

El *hardware* para controlar la memoria se hace más complejo, pero el esquema general tiene la ventaja de que el programador no tiene que tomar en cuenta la forma en que se administra la memoria, dado que las unidades de su diseño están contiguas, aunque el programa, como un todo, esté disperso en diferentes segmentos.

SEGMENTO	INICIO	LONGITUD
	...	

Figura III.9. Tabla de segmentos. Fuente elaboración propia.

En lugar de una tabla de página, se tiene una **tabla de segmentos** que tiene tres campos al menos (figura III.9):

* El primer campo (**Segmento**) contiene el número del segmento.
* El segundo campo (**Inicio**) contiene la dirección de comienzo del segmento.
* El tercer campo (**Tamaño**) contiene la longitud del segmento.

Cada vez que llega una dirección en la forma <s, d>, siendo s el número del segmento y d el desplazamiento dentro del segmento, el SO (con el apoyo del *hardware*) realiza las siguientes acciones:

1. Busca, en el campo Segmento de la tabla de segmentos, el segmento s, sea esta la localización S_i dentro de la tabla.
2. Si el desplazamiento d está en rango ($0 \leq d \leq S_i$.Longitud), es decir que si el desplazamiento de la dirección (d) dentro del segmento es mayor o igual que cero y menor o igual que el tamaño del segmento, entonces:
   a. Calcula la dirección solicitada para lo cual se suma la dirección base del segmento (S_i.Inicio) con el desplazamiento (d): S_i.Inicio + d
3. Si no se cumple $0 \leq d \leq S_i$.Longitud, hay un error de direccionamiento y el SO tiene que intervenir (ocurre una trampa al SO).

III.3. MEMORIA VIRTUAL

Como ya se ha visto, existen muchas estrategias para el manejo de la memoria, y todas ellas persiguen el fin común de mantener la mayor cantidad posible de procesos cargados en memoria, de modo que se puedan ejecutar simultáneamente.

En un sistema multiprogramado, es imposible mantener todos los procesos cargados en memoria; sin embargo, solo los procesos que estén en memoria compiten por el procesador. ¿Cómo se resuelve esta aparente contradicción?

Para resolver el problema, se usa la **memoria virtual**; el resto del capítulo explica los detalles de esta técnica.

La idea de la memoria virtual parte de la observación de que en un sistema con un solo procesador solo se ejecuta una instrucción en cada instante de tiempo, de modo que, en un caso extremo, el SO podría cargar en memoria instrucción por instrucción del proceso seleccionado para después ejecutarla. Resulta obvio que dicha consideración es muy extrema y haría que el proceso se ejecutara muy lentamente y, en general, no resulta nada práctica, por eso la realidad es menos drástica.

Otra consideración que se toma en cuenta es que el código de un programa tiene determinadas partes que se ejecutan muy rara vez (por ejemplo, algunas excepciones, tales como el tratamiento de errores).

Es bueno destacar los beneficios que puede reportar el hecho de que un proceso pueda ejecutarse aunque no esté completamente cargado en memoria.

* La primera e importante ventaja, es que el proceso podrá ejecutarse aunque la memoria de la máquina sea menor que sus necesidades. Esto quiere decir que el programador no estará restringido por la capacidad de la memoria y no tendrá que lidiar con algoritmos especiales para resolver esta situación si se le presentara (como sucedía en el pasado).

* Debido a que los procesos que se ejecutan tendrán requerimientos menores de memoria, más procesos podrán ejecutarse "a la vez". Esto da como resultado que se incremente el uso de la CPU, lo cual siempre ha sido uno de los objetivos que debe cubrir el SO.

III.3.1 Solapamientos (*overlays*)

Esta técnica prácticamente no se usa, pero su estudio ayuda a entender los principios básicos de la memoria virtual. El solapamiento necesita un análisis del proceso para determinar cuáles partes pueden dividirse en unidades lógicas que pueden trabajar de forma independiente.

Una parte de la memoria física se declara como zona de solapamiento (*overlay*); en esa zona, se cargarán las partes independientes cada vez que se necesiten, de modo que solo una de ellas estará en memoria en cualquier instante de tiempo.

La pregunta clave es:

* ¿De qué tamaño se declara la zona de solapamiento? La respuesta es obvia, tendrá que ser mayor o igual que la mayor de las partes independientes que se cargarán en ella.

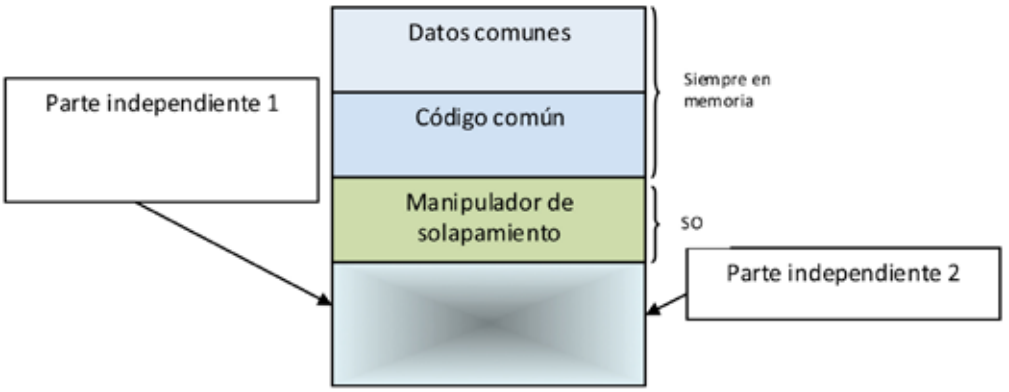


Figura III.10. Esquema de solapamiento típico. Fuente elaboración propia.

Entonces, en este esquema es necesario tener espacio para dos partes que estarán en memoria, es decir que deberán existir dos zonas de memoria (figura III.10):

* Una zona que contiene los datos y códigos comunes a las dos partes, que permanece constantemente en memoria.
* Otra zona para el solapamiento que será igual a la longitud de la mayor de las partes independientes (parte independiente 1 en la figura III.10).

Para que el proceso comience su ejecución, primero se cargará la parte común y una de las partes independientes en la zona de solapamiento, cuando se necesite la otra parte independiente, el manipulador de solapamiento se encargará de cargarla en la zona de solapamiento. Las partes independientes pueden entrar y salir de la zona de solapamiento durante el transcurso de la ejecución del proceso y hasta su terminación.

III.3.2 Demanda de página

La **demanda de página** se basa en el esquema paginado y, en sí, es el paginado más un sistema de intercambio (*swapping*) de páginas desde y hacia la memoria.

Antes de entrar en detalles de la técnica como tal, resulta imprescindible definir algunos conceptos:

Los recursos de *hardware* que se usan para este tipo de manejo de memoria son:

* El **equipo de resguardo** (*backing store*). Se usa para tener una copia del contenido de la memoria de cada proceso. Con ese fin, se puede usar: un disco duro destinado solo a eso, una partición dentro del disco duro (como lo hace la familia Unix), o un archivo especial que reside en un disco duro (como lo implementa la familia de SO Windows).
* La memoria.

La idea central es no cargar en memoria los programas completos, solo se cargarán algunas páginas, las cuales se mantendrán en memoria hasta que sea necesario desalojarlas para cargar otra parte.

Cada página se cargará en un **marco de página**; la tabla de página permitirá deducir cuáles páginas están en memoria y cuáles no, así como otras cosas que ayudarán a evitar el exceso de entradas y salidas de las páginas a memoria (se conoce como tráfico de páginas o *trashing*).

La solicitud de una página se conoce como **demanda de página**. A continuación, se presenta la primera versión del algoritmo para satisfacer estas solicitudes.

```
Algoritmo de demanda de página. Versión 1
Si la página solicitada está referenciada en la tabla de página
  Entonces
    /* Comentario A
       No es necesario cargar la página porque ya estaba en memoria.
       De acuerdo al algoritmo de reemplazamiento se pueden hacer
       algunas acciones sobre la tabla de página, por ejemplo: marcar
       el momento (tiempo) en que se usó la página, incrementar un
       contador de uso, etc.*/
  En caso contrario                    //Hay una falta de página
    Si hay algún marco de página (frame) libre
      Entonces
        Cargar la página en marco de página seleccionado.
        Inicializar los campos mencionados en comentario A.
  //Acciones generales
  Actualizar la tabla de página.
  Calcular la dirección física.
  Retornar la dirección calculada.
Fin
```

Cada vez que se hace una demanda de página, el SO (auxiliado por el *hardware*) usa la tabla de página para calcular la dirección física donde está cargada la página solicitada. Cuando la página solicitada no está en memoria, ocurre una interrupción que se denomina **falta** o **fallo de página**; en ese momento es necesario ejecutar las acciones siguientes:

1. Cargar la página solicitada, que está en el equipo de resguardo, hacia la memoria.
2. Actualizar la tabla de página.
3. Satisfacer la demanda de página que había quedado pendiente.

Una forma de hacer notar que una página dada no está en memoria es agregando un **bit de presencia** asociado a cada entrada de la tabla de página. El bit tendrá el valor 1 si la página está en memoria y -1 si no está (esto es solo un convenio y puede ser cualquier otro valor).

Es importante destacar que la falta de página es una interrupción al proceso que se está ejecutando, lo cual trae como resultado que el proceso solicitante se detenga hasta que se reponga de esa interrupción. El SO debe guardar los registros del procesador para que después el proceso pueda continuar exactamente por el lugar donde iba. También es necesario guardar la tabla de página, todos esos datos se guardan en el PCB del proceso (discutido en el capítulo I).

La versión 1 del algoritmo para atender la demanda de página presentado previamente supone que siempre hay un marco de página libre; de ese modo, siempre es posible cargar la página hacia la memoria y continuar el curso del proceso (puede que no sea en ese instante, debido a que se le asigne el procesador a otro proceso), pero:

* ¿Qué sucede si ningún marco de página está libre?

En ese caso, ocurre un **reemplazamiento de página**, lo cual significa que el SO debe elegir una página para ser reemplazada y sustituida por la página demandada, es decir, se produce un **intercambio de páginas** (*swapping*): la página que ha sido elegida para ser reemplazada se denomina **página víctima** y debe guardarse en el equipo de resguardo, mientras que la página que provocó la falta de página se trae a memoria.

Puede notarse que no siempre es necesario salvar la página víctima hacia el equipo de resguardo, dado que este siempre tiene una copia de ella. Entonces la pregunta es:

* ¿Cuándo debe salvarse la página víctima?

Resulta claro que solo es necesaria esta acción cuando dicha página haya sido modificada durante su estancia en la memoria interna. Para facilitar esta tarea, se agrega un nuevo campo a la tabla de página que se denominará **bit de suciedad** (*dirty bit*).

La idea del bit de suciedad (*dirty bit*) es sencilla: cada vez que una página se carga hacia memoria, viene con su bit de suciedad **limpio**; si se modifica mientras esté en memoria, habrá que salvarla cuando se reemplace; si no se **ensucia** (no se modifica), no será necesaria esa acción dado que existe una copia de ella en el equipo auxiliar de resguardo (*backing store*).

La versión 1 del algoritmo para atender la demanda de página debe modificase para incluirle estas últimas consideraciones; seguidamente se presenta la nueva versión.

```
Algoritmo de demanda de página. Versión 2
Si la página solicitada está referenciada en la tabla de página
  Entonces
  /* Es válido el comentario A de la versión 1*/
  En caso contrario                    //Hay una falta de página
    Si hay algún marco de página (frame) libre
      Entonces
        Cargar la página en marco de página seleccionado.
      En caso contrario                //Hay un reemplazamiento de página
        Seleccionar una página víctima.
        Si la página víctima está sucia
          Copiarla hacia el equipo de resguardo.
        Copiar nueva página hacia el marco seleccionado.
  //Acciones generales
  Actualizar la tabla de página.
  Calcular dirección física.
  Retornar la dirección calculada.
Fin
```

Para implementar la demanda de página, se pueden seguir las dos técnicas siguientes:

* Demanda de página pura. Consiste en no cargar las páginas hasta que no hayan sido solicitadas. En este caso, cuando se ordena la ejecución de un proceso, solo se carga la página donde está el punto de entrada del programa (*entry point*[40]), y las demás páginas se irán cargando cuando sean solicitadas por primera vez.
* Prepaginado. Consiste en cargar inicialmente un conjunto de páginas. Para hacer esta acción, se puede tomar en cuenta un concepto de vecindad, donde se espera que se use la página donde está el punto de entrada del programa más algunas páginas vecinas (eso no tiene que ser cierto en la realidad, es solo una forma de tomar decisiones).

III.3.3 Demanda de segmentos

La idea de la demanda de segmentos es igual a la de demanda de páginas, pero en este caso resulta mucho más compleja.

Un esquema segmentando ve la memoria como un gran espacio que inicialmente está libre; la entrada y la salida de los segmentos a memoria va dejando "huecos" de diferentes longitudes, es decir que la fragmenta.

Deben tomarse en cuenta las siguientes consideraciones:

1. Cuando se desea traer un segmento a la memoria (cuando ocurre una **falta de segmento**), no basta con que haya un espacio libre; en ese caso, habrá que comprobar si alguno es lo suficientemente grande para alojar el segmento.
2. Cuando existe la necesidad de reemplazar un segmento por otro, no se puede escoger como víctima a cualquier segmento, dado que solo serán candidatos aquellos que tengan una longitud mayor o igual que el segmento que se traerá a la memoria.
3. Además, en ambos casos, se debe considerar el hecho de que el nuevo segmento no tiene que ocupar todo el espacio que había libre o que se libere y, por tanto, será necesario actualizar la lista de segmentos libres, tomando en cuenta adicionalmente que si el pedazo de segmento que sobra se va a liberar contiguo a otro "hueco" de memoria libre que existía antes, habrá que combinar ambos espacios en uno solo.

Antes de presentar un algoritmo para la demanda de segmento, se establece la siguiente notación:

* TSL (tabla de segmentos libres). Es una estructura de datos donde se referencian los segmentos libres; su forma general se aprecia en la figura III.11, aunque puede ser otra, por ejemplo, un mapa de bits.
* S_i. Se refiere al segmento con el número i.

40 El punto de entrada es el lugar por donde comienza la ejecución del programa.

* TS (tabla de segmentos). Cada proceso tiene una TS, la cual es una estructura de datos que contiene los datos de localización de sus segmentos; su forma general se aprecia en la figura III.11.
* Para un segmento S_i: $|S_i|$ denota su longitud.
* La función long(T) devuelve la longitud de una tabla, expresada como su último índice, es decir que si una tabla T tiene 3 elementos, long(T) es 3.

TSL	
Dirección Base (DB)	Longitud (L)
	...

TS		
Número del segmento (N)	Dirección Base (DB)	Longitud (L)

Figura III.11. Tabla de segmentos libres (TSL) y tabla de segmentos (TS). Fuente elaboración propia.

La TS no contiene espacios vacíos, como en la tabla de página, debido a que los segmentos se referencian por la dirección base y no por la posición en la TS como ocurre en la tabla de página. Las referencias a segmentos nuevos siempre se ponen al final de la TS.

Algoritmo general para la demanda de segmentos DS

En el algoritmo que se describe a continuación, debe observarse que:

* Cuando se escoge un segmento víctima, para cargar un nuevo segmento en esa dirección no es necesario actualizar el campo DB de TS, debido a que la dirección de inicio del nuevo segmento será la misma del segmento víctima.
* Los campos N (número del segmento) y L (longitud del segmento) deberán actualizarse.
* Normalmente, la longitud de un segmento que se carga en la posición de otro no coincide con la longitud del segmento víctima y sobrará algún espacio que habrá que poner como libre en la TSL.

```
Algoritmo de demanda de segmento
DS(Si, L)  // Se solicita el segmento Si de Longitud L
  Si Si está en TS
    Entonces
  Retornar(TS[i].DB);
    En caso contrario                          // Hay una falta de segmento
      {
        Para(j = 0; j < long(TSL); j++)        // Buscar un segmento libre S tal que: |S| ≥ |L|
          Si |L| ≤ TSL[j].L
            {
              //Se inserta una entrada nueva al final de la tabla de segmentos
              TS[long(TS)].N      = i;         // El segmento que se cargará es el número i
              TS[long(TS)].DB     = TSL[j].DB; /*La dirección de inicio del nuevo segmento
                                                 es la dirección de inicio del segmento libre escogido*/
              TS[long(TS)].L      = L;         // La longitud del segmento se recibió como parámetro
              Si TSL[j].L > |L|                /* Si la longitud del nuevo segmento no ocupa todo
                                                 el espacio del segmento libre */
                Actualizar(TSL);               // Es necesario actualizar la TSL
              Cargar(Si, TSL[j].DB);           // Cargar segmento i a partir de TSL[j].DB
              Retornar(TS[j].DB);              //Retornar la dirección del nuevo segmento
          }
        /* Si no existe segmento libre, es necesario buscar un segmento víctima */
        Para(j = 0; j < long(TS); j++)         // Buscar un segmento víctima (usa algún algoritmo)
          Si |L| ≤ TS[j].L                     /*La longitud del segmento víctima tiene que ser
                                                 mayor o igual que el segmento i*/
            {
              Si TS[j].N está sucio            // Si el segmento víctima se ha modificado
                Guardar TS[j].N en BS;         //hay que guardarlo en el equipo de resguardo
              Cargar (Si, TS[j].DB);           //Cargar segmento i a partir de TSL[j].DB
              // Sustituir la entrada j en la TS
              TS[j].N = i,                     // El segmento es ahora el i
              TS[j].L = L                      // Su longitud es la pasada por parámetro
              Si TS[j].L > |L|                 /* Si la longitud del segmento víctima es mayor que
                                                 la del segmento i sobra un espacio, actualizar TSL */
                Actualizar(TSL);
              Retornar(TS[j].DB);
            }
```

III.4 ALGORITMOS DE REEMPLAZAMIENTO

Existen diversos algoritmos para escoger la parte de un proceso que deberá reemplazarse en la memoria. En esta sección, se hace referencia (por simplicidad) a los algoritmos de reemplazamiento de páginas, pero pueden también ser ajustados al reemplazamiento de segmentos (tomando en cuenta que el segmento escogido tiene que ser mayor o igual que el nuevo segmento a cargar). Solo se estudian tres algoritmos.

El primero que llega es el primero en salir (First In First Out - FIFO)

Es el más simple de los algoritmos de reemplazamiento y consiste en tomar como página víctima a la más vieja, entendida como tal la que lleve más tiempo en memoria. La longevidad de la página se puede conocer guardando la hora en que se carga a memoria.

Como lo que interesa conocer es cuál es la más vieja y no su edad, puede usarse una cola tipo FIFO, en la que se insertan las páginas al final de la cola cuando entran a memoria, así la más vieja siempre estará en la cabeza de la cola.

El algoritmo FIFO es fácil de comprender y programar, pero su rendimiento no es bueno (muchas veces) y sufre de un fenómeno que consiste en que, en ocasiones, al

aumentar la cantidad de marcos de página que cada proceso puede tener en memoria, se produce una mayor cantidad de faltas de páginas. Este comportamiento va en contra de la "lógica" de cualquier persona y se conoce como **anomalía de Beladay** en honor de su descubridor.

Reemplazamiento óptimo

Un algoritmo de reemplazamiento óptimo no puede sufrir la anomalía de Beladay, y debe provocar la menor cantidad de faltas de páginas de todos los algoritmos.

El algoritmo óptimo toma como víctima a la página que más va a demorar en ser referenciada; es decir, este algoritmo **mira hacia el futuro** y escoge para ser sustituida una página que va a demorar mucho (más que todas) en ser referenciada, lo cual garantiza que el tiempo en que se producirá la siguiente falta de página será el más largo de todos.

La mala noticia es que resulta totalmente imposible predecir el comportamiento de un proceso y, por tanto, el algoritmo solo se usa para hacer comparaciones: "mientras más se acerque un algoritmo al óptimo, mejor será"; también puede hacerse alguna aproximación a él con base en estadísticas de comportamientos anteriores.

La página menos recientemente usada (Least Recently Used - LRU)

Mientras el algoritmo óptimo fracasa tratando de predecir el futuro, el algoritmo LRU **mira al pasado** y supone que lo que ha sucedido en tiempos anteriores puede ser un reflejo de lo que ocurrirá en adelante.

En esencia, LRU toma como víctima a la página que lleva más tiempo sin ser referenciada y asume la actitud optimista de que si la sustituye, no la va a necesitar en un tiempo considerable.

Este algoritmo necesita que a cada página se asocie el tiempo en que fue usada (debe observarse que no es cuando fue cargada como ocurre en FIFO). LRU se usa mucho y estadísticamente muestra un buen comportamiento, pero necesita apoyo del *hardware*.

III.5 RESUMEN DEL CAPÍTULO

La memoria es uno de los recursos fundamentales de cualquier sistema de cómputo y se puede ver como un gran arreglo de bytes o palabras que tienen una dirección asociada.

La idea de la multiprogramación es tener varios procesos cargados en memoria para poder intercambiar el procesador entre ellos y dar la sensación de que se están ejecutando a la vez.

Para lograr la multiprogramación, es necesario administrar la memoria de forma que sea suficiente para cargar los procesos.

Para que un proceso se ejecute, no es necesario que todos sus datos y códigos estén en memoria. En esa observación se basan las técnicas de memoria virtual, las cuales implementan algunas ideas para tener solo una parte de cada proceso en memoria y cargar el resto cuando sea necesario, muchas veces sustituyendo las que estaban antes.

Existen diversas implementaciones de la memoria virtual; las más usadas son el paginado y la segmentación.

Tanto el paginado como la segmentación se basan en algoritmos que permiten escoger, entre las partes cargadas en memoria, aquellas que serán sustituidas por otras. Las partes escogidas para ser sustituidas por otras reciben el nombre de **víctimas** y el SO debe tratar de usar algoritmos que no provoquen un tráfico excesivo entre la memoria y el equipo auxiliar, debido a que durante ese tiempo los procesos involucrados deben detenerse.

Los sistemas paginados dividen la memoria física en unidades del mismo tamaño y longitud fija, conocidas como marcos de página. La memoria lógica de cada proceso también se divide en unidades del mismo tamaño, denominadas páginas, de forma que cualquier página cabe dentro de cualquier marco de página. El paginado usa una estructura de datos, llamada tabla de página, que le permite conocer en qué marcos de página están cargadas las páginas.

Los sistemas segmentados ven la memoria en forma de unidades lógicas asociadas a concepciones de programación, tales como funciones, estructuras de datos diversas, procedimientos, etc. Esta concepción hace que los segmentos sean de diferentes longitudes y que la estructura de datos que se usa para localizarlos, la tabla de segmentos, tenga que agregar campos que especifiquen la dirección de inicio de cada segmento y sus longitudes.

Los sistemas segmentados sufren de fragmentación externa, mientras que los sistemas paginados padecen de fragmentación interna.

III.6. EJERCICIOS

1. Explique en qué circunstancias ocurre la falta de página y describa las acciones que hace el SO en ese caso (expréselas en forma de un algoritmo).
2. Asuma que en un sistema con m marcos de página (*frames*), inicialmente todos vacíos, un proceso P tiene una cadena de referencias a páginas de longitud *l* con n referencias a páginas distintas. Sin importar el algoritmo:
   a. ¿Cuál será la menor cantidad de faltas de páginas que puede ocurrir?
   b. ¿Cuál será la mayor cantidad de faltas de páginas que puede ocurrir?
3. Escriba un algoritmo, con todos los detalles, para atender la demanda de página usando un algoritmo de reemplazamiento específico.
4. Escriba un algoritmo, con todos los detalles, para atender la demanda de segmento usando un algoritmo de reemplazamiento específico.
5. Haga una valoración crítica acerca de los algoritmos que se analizan en este libro y otros que deberá buscar en bibliografías complementarias.
6. Explique el concepto de anomalía de Beladay.
   a. Para un algoritmo específico ponga un ejemplo de una cadena de referencias a páginas, donde se muestre la ocurrencia de la anomalía de Beladay. Considere un sistema paginado puro.
   b. Haga el mismo análisis para un sistema prepaginado.

III.7. BIBLIOGRAFÍA CONSULTADA

Abraham, S., Baer- Galvin, P., & Gagne, G. (2013). *Operating systems concepts* (9.ª ed.). Nueva Jersey: John Wiley & Sons.

Belady, L., Robert A. N., & Shedler, G. (1969). An anomaly in space-time characteristics of certain programs running in a paging machine. *Communications of the ACM*, *12*(6), 349-353.

Dhamdhere, D. M. (2008). *Operating systems. A concept based aproach.* Nueva York: Mc-Graw-Hill Education.

Elmasri, R., Carrick, A., & Levine, D. (2009). *Operating systems: A spiral approach.* Nueva York: McGraw-Hill.

Harvey, M., Deitel, P., & Choffnes, D. (2003). *Operating systems* (3.ª ed.). Nueva Jersey: Prentice Hall.

Koranga, M., & Koranga, N. (2014). Analysis on page replacement algorithms with variable number. *International Journal of Advanced Research in Computer Science and Software Engineering*, *4(7)*, 403-411.

Sadeh, E. (1975). An analysis of the performance of the page fault frequency (PFF) replacement algorithm. *SOSP '75. Fifth ACM Symposium On Operating Systems Principles*, *9(5)*, 6-13.

Stallings, W. (2014). *Operating systems: Internals and design principles* (8.ª ed.). Nueva York: Pearson.

Tanenbaum, A. (2006). *Operating systems: Design and implementation* (3.ª ed.). *Chapter 4.* Nueva Jersey: Prentice Hall.

Tanenbaum, A., & Bos, H. (2014). *Modern operating systems* (4.ª ed.). Nueva York: Pearson.

Segunda parte

Caso de estudio. Unix y sus descendientes

Second part

Case study. UNIX and its descendants

CAPÍTULO IV

Generalidades del SO Unix y sus descendientes

RESUMEN

Este capítulo es un caso de estudio acerca de los sistemas operativos tipo UNIX. El capítulo comienza mostrando cómo debe usarse la documentación en línea a través del comando man, para después ofrecer una panorámica de los intérpretes de comandos o Shell, así como la forma de trabajo con los procesos y los comandos. A continuación, se enfatiza acerca de la redirección de entrada/salida, las expresiones regulares y su uso con diferentes propósitos, las tuberías y los filtros, así como la utilización de los patrones de archivos. El capítulo finaliza con un resumen y una sección de ejercicios propuestos.

Palabras clave: UNIX, comandos, filtros, tuberías, redirección, expresiones regulares.

¿Cómo citar este capítulo? / How to cite this chapter?

Lezcano-Brito, M. G. (2017). Generalidades del SO Unix y sus descendientes. En *Fundamentos de sistemas operativos. Entornos de trabajo* (pp. 131-170). Bogotá: Ediciones Universidad Cooperativa de Colombia.

CHAPTER IV

Overview of UNIX OS and its descendants

ABSTRACT

This chapter is a case study of UNIX-like operating systems. The chapter begins by showing how to use online documentation through the *man* command and then provides an overview of command interpreters or shell, as well as how to work with processes and commands. Next, emphasis is placed on input/output redirection, regular expressions and their use for different purposes, pipelines and filters, and the use of file patterns. The chapter ends with a summary and a section of proposed exercises.

Keywords: UNIX, commands, filters, pipelines, redirection, regular expressions.

Una vez analizados los aspectos generales relacionados con el diseño y la implementación de los SO, se tienen las herramientas necesarias para abordar las particularidades de algún SO, lo que incluye el entorno de programación.

Esta segunda sección está dedicada a los SO de la familia Unix[41], se trata de ser lo menos particular posible para que los temas discutidos se puedan aplicar a cualquiera de los SO de esta gran familia.

El capítulo IV, en particular, abarca temas generales de los SO Unix o tipo Unix, y es una especie de preámbulo para el capítulo V que trata los aspectos relacionados con la programación usando herramientas propias del SO.

IV.1 DOCUMENTACIÓN EN LÍNEA

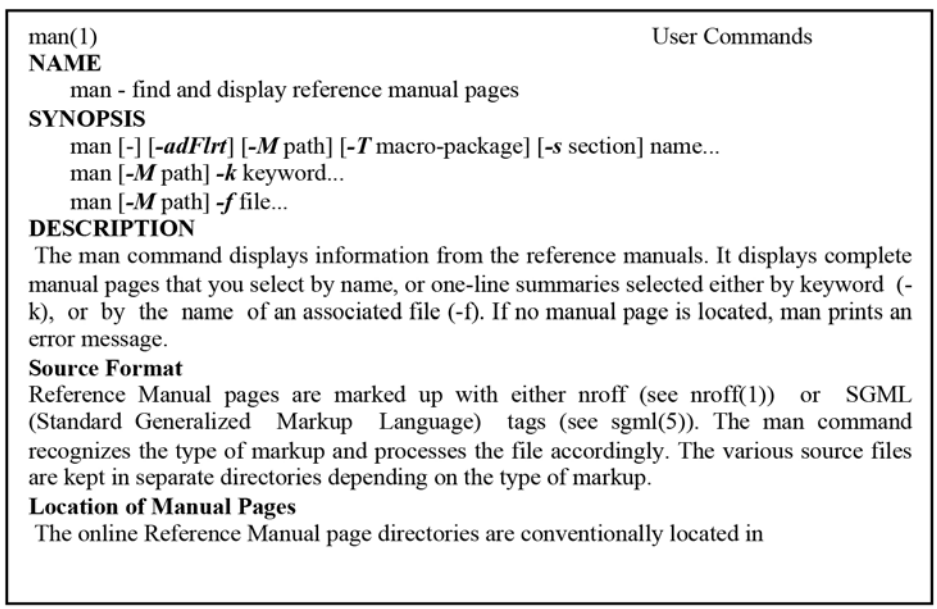
Existen muchos y buenos libros generales acerca del SO Unix, pero para analizar las particularidades en un entorno de la familia Unix, se debe siempre usar la ayuda que se instala junto con el SO.

La ayuda de los SO de la familia Unix se obtiene usando el comando externo man (manual). Si se teclea ese comando desde una terminal cualquiera, se verá algo como lo mostrado en la figura IV.1. La ayuda se divide en nueve secciones:

1. En la primera sección, se explican los comandos del SO, por ejemplo: cat, clear, ls, etc.
2. En la segunda sección, se ofrecen los detalles de las llamadas al sistema, las cuales tienen la forma de una invocación normal a una función C, por ejemplo: fork(), getpid(), signal(), etc.
3. La tercera sección explica las llamadas a funciones que están dentro de distintas bibliotecas de programas, por ejemplo: execl(), pthread_create, atoi(), etc.
4. La cuarta sección del manual se dedica a archivos especiales que, muchas veces, están en el directorio dev (*devices*), por ejemplo: full, ram, port, etc.
5. La quinta sección brinda ayuda acerca de archivos de formatos y convenciones, por ejemplo: passwd, core, crontab, etc.

41 En inglés se usa la expresión ***Unix-like*** para referir un sistema que se comporta de manera similar a Unix y en español se usará la frase "sistema operativo tipo Unix".

6. En la sexta sección, se ofrecen detalles de los juegos; por ejemplo, intro ofrece una introducción a los juegos, pero los detalles dependerán de cada fabricante.
7. La sección 7 es una miscelánea acerca de diferentes tópicos que no pueden agruparse en otras categorías. Algunas son macros[42], como man (no confundir con la ayuda que está en la sección 1 y también se llama man); así mismo, se explican protocolos como arp, etc.
8. La sección 8 está dedicada a comandos que usan los administradores, por ejemplo: arp, mkfs, mount, etc.
9. Por último, la sección 9 se encarga de las rutinas del núcleo (*kernel*) del SO.



man(1) User Commands

NAME

man - find and display reference manual pages

SYNOPSIS

man [-] [***-adFlrt***] [***-M*** path] [***-T*** macro-package] [***-s*** section] name...
man [***-M*** path] ***-k*** keyword...
man [***-M*** path] ***-f*** file...

DESCRIPTION

The man command displays information from the reference manuals. It displays complete manual pages that you select by name, or one-line summaries selected either by keyword (-k), or by the name of an associated file (-f). If no manual page is located, man prints an error message.

Source Format

Reference Manual pages are marked up with either nroff (see nroff(1)) or SGML (Standard Generalized Markup Language) tags (see sgml(5)). The man command recognizes the type of markup and processes the file accordingly. The various source files are kept in separate directories depending on the type of markup.

Location of Manual Pages

The online Reference Manual page directories are conventionally located in

Figura IV.1. Mostrando la ayuda en un entorno Ubuntu.

Un aspecto importante al usar la ayuda es que el comando man busca la información a partir de la primera sección, lo que significa que si hay dos asuntos con igual nombre y no se especifica en qué sección se debe buscar, siempre saldrá la primera información que se encuentre. Por ejemplo, si se teclea la orden: man kill, saldrá información acerca del comando kill que está localizada en la sección 1 del manual (en el extremo superior izquierdo de la pantalla de ayuda se podrá ver el número de la sección). Sin embargo, si se deseara saber acerca de la llamada al sistema de igual nombre, será necesario incluir el número de la sección; como las llamadas al sistema están en la sección 2, el comando adecuado será man -s 2 kill, donde s (*section*) es una opción que permite especificar la sección en la que se debe buscar.

42 Una macro o macroinstrucción es un conjunto de instrucciones que se agrupan bajo un nombre, de manera que se puede invocar desde diferentes lugares sin necesidad de repetir las instrucciones que lo conforman. No debe confundirse con el concepto de función.

IV.2 ASPECTOS GENERALES

Unix es un SO con muchos años de existencia; su primera versión emergió de los laboratorios Bell[43] (que era una parte de AT&T) en 1969. El SO se hizo específicamente para explotar una mini computadora PDP-7 de aquella época y sus principales creadores fueron Ken Thompson y Dennis Ritchie.

La primera versión del sistema se escribió en lenguaje ensamblador, y en la tercera versión, la mayoría del código estaba escrito en el lenguaje C, que fue desarrollado con ese fin, de ahí que las historias del C y del UNIX estén estrechamente unidas.

Hoy en día, existe una gran variedad de versiones de Unix que se ejecutan sobre diferentes tipos de plataformas. Los SO de esta familia se han extendido por todo el mundo, alcanzando una gran popularidad y aceptación. Las versiones de Unix tienen sus diferencias, pero en general siguen una línea común, lo que hace que no sea tan difícil desarrollar habilidades en una de ellas y después pasarse a otra.

Una rama importante del tronco común Unix es la de los SO Linux. El proyecto surgió en 1993 como la idea de un estudiante llamado Linus Torvalds. Linus cursaba la carrera de informática en la Universidad de Helsinki y tomó las ideas del núcleo del SO Minix[44].

Para no tener que hacer muchas especificaciones, cuando se diga Unix en este capítulo se estará hablando de cualquier SO que sea descendiente de Unix.

Unix como SO multiusuario

Unix es un SO multitarea y multiusuario. El hecho de ser multiusuario permite que se conecten a él muchas personas desde diferentes lugares. Esas conexiones se pueden hacer a través de **terminales**.

Una terminal básica o tonta (son inusuales hoy en día) no es más que un equipo compuesto de un teclado y una pantalla, aunque puede tener algunos otros componentes.

Las terminales tontas solo tienen capacidad de trabajar con textos, pero existen terminales más complejas como las **terminales X** que permiten trabajar en modo gráfico. Además, una computadora puede emular una terminal.

Cuando se instala un SO de la familia Unix para que sirva a diferentes terminales, la computadora donde se instala el SO recibe el nombre de **anfitrión** y maneja todo lo que se hace en las demás terminales; por ejemplo, el simple hecho de oprimir una teclea hace que ese carácter se envíe hasta la computadora anfitrión, la cual lo devuelve (se dice que "hace eco de él") para mostrarlo en la pantalla de la terminal. Como todo eso ocurre rápido, puede ser que no se note.

La figura IV.2 muestra la idea de un SO Unix manejando diferentes terminales. Las tareas de administración se hacen desde la consola, que no es más que el teclado y el monitor de la computadora anfitriona (la que tiene instalado el SO).

43 https://www.bell-labs.com/

44 El SO Minix lo creó, en 1987, Andrew S. Tanenbaum (Vrije Universiteit de Ámsterdam). Está especialmente concebido para enseñar aspectos del diseño de sistemas operativos y se distribuye libremente junto a su código fuente.

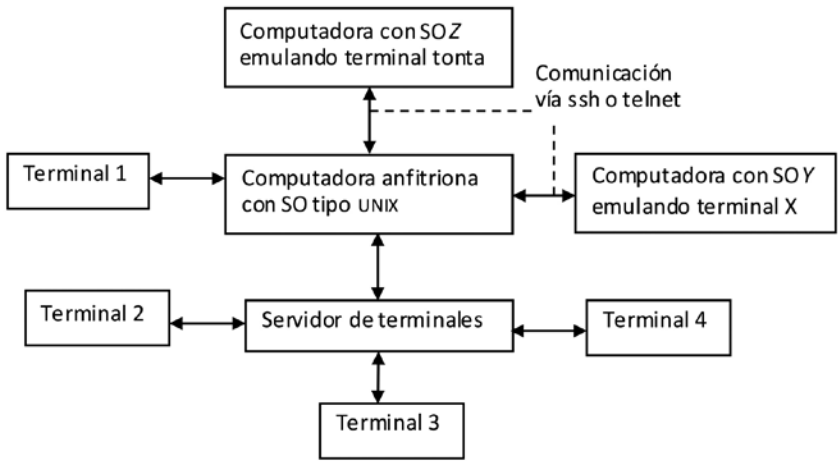


Figura IV.2. Unix como SO multiusuario.

La conexión de las terminales con la computadora anfitriona es directa, pero la conexión desde otra computadora se puede hacer a través de diferentes protocolos de conexión, por ejemplo: telnet[45], que no es una conexión segura, o ssh (*security shell*)[46], que es una conexión segura.

Los siguientes ejemplos muestran la forma de conexión a un SO Unix desde el intérprete de comandos del SO Windows (observe que el símbolo **C:>,** lo pone Windows). En el primer y el tercer ejemplo, se usa el número ip de la máquina Unix y en el segundo y el cuarto se usa su nombre:

```
C:>telnet 13.12.2.23
C:>telnet UnixSystem
C:>ssh 13.12.2.23
C:>ssh UnixSystem
C:>
```

Una variante más cómoda es usar un programa que facilite la conexión; ese es el caso del programa **putty** que se puede bajar libremente de Internet. Este programa también permite definir algunas especificaciones adicionales para la conexión.

Una vez establecida la conexión, el sistema Unix responderá solicitando los datos: **nombre de usuario** y **contraseña** para autenticar que el usuario tiene los permisos adecuados y asignarle los derechos que se le dieron cuando se creó su cuenta.

En el diálogo siguiente, el usuario **mlezcano** entró al sistema, para lo cual tecleó su identificación o *login* (mlezcano) y su palabra clave (*password*), de la cual no se conoce ni siquiera cuántos caracteres tiene debido a que esa acción no produce eco, esta es una forma adecuada para preservar la clave de mirones.

45 Permite conectar terminales y aplicaciones, proporciona reglas básicas para conectar un cliente con un intérprete de comandos del lado del servidor.

46 Ofrece una comunicación segura entre dos sistemas, para lo cual encripta la sesión de conexión y usa una arquitectura cliente/servidor.

```
login as: mlezcano
Password:
Linux UnixSystem 2.6.8-2-686 #1 Tue Aug 16 13:22:48 UTC 2015 i686
Bienvenidos al Servidor de Archivos DEBIAN
Nombre: UnixSystem
IP: 13.12.2.23
Last login: Wed Feb 13 10:54:43 2016 from myMachine
mlezcano@ UnixSystem:~$
```

El SO debe autenticar si es válida la combinación (*login*, *password*). En este caso, la reconoció y se presentó con un mensaje que informa desde dónde se conectó ese usuario la última vez y también la fecha en que lo hizo. Es bueno verificar esos datos para saber si alguien ha hecho una conexión no autorizada a nuestro nombre.

Saliendo del sistema

Dependiendo del sistema Unix que se use, existen diferentes formas para salir y terminar una sesión de trabajo desde una terminal (nunca debe dejarse una sesión abierta); algunas son:

* **<ctrl d>**, o sea, las teclas control y d al mismo tiempo. Al teclear esta combinación de teclas, se envía una señal de fin de archivo al Shell, el cual la interpreta como una orden para cerrar la sesión.
* **logout**
* **exit**

La forma de conectarse directamente a un SO de la familia Unix dependerá del sistema particular, pero en todo caso se tendrá que pasar por el protocolo de identificarse y escribir la palabra clave.

Existe un usuario especial dentro de los SO Unix que tiene el nombre root, él es el superusuario y tiene todos los privilegios; en particular, está facultado para apagar la computadora de forma adecuada, para lo cual deberá ejecutar el comando shutdown.

El comando shutdown hace una salida del sistema en forma segura; antes de salir, les notifica esa acción a todos los usuarios e informa el tiempo que esperará para comenzar a hacer las acciones asociadas a la salida. El usuario facultado para hacer shutdown puede especificar ese tiempo.

Organización del sistema de archivo

En el capítulo II, se ofreció una idea general acerca de los sistemas de archivos de la familia Unix, pero es importante volver sobre algunos conceptos y ampliar otros.

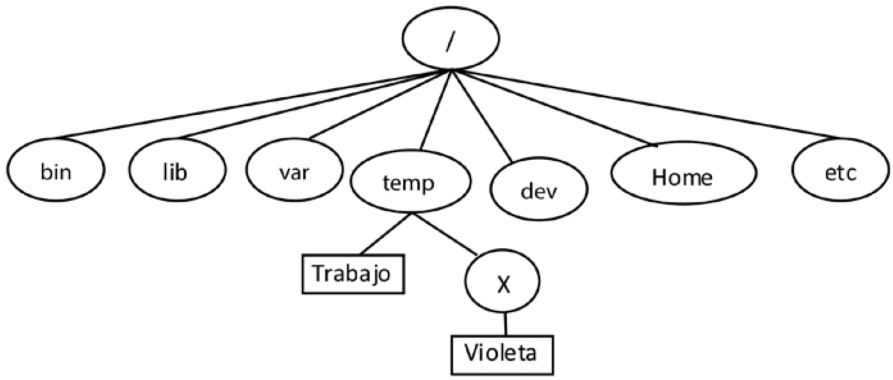


Figura IV.3. Estructura de un directorio en forma de árbol invertido.

En Unix, los archivos están organizados en **directorios jerárquicos** (figura IV.3), en forma de árbol invertido (la raíz arriba y las hojas abajo). Esos directorios son a su vez archivos especiales que contienen información acerca de otros archivos.

En la figura IV.3 el símbolo / (diagonal o *slash*) representa el directorio raíz (*root*), mientras que bin, lib, var, temp, etc., son hijos del directorio raíz.

Obsérvese que el directorio temp tiene dos hijos:

* El primero, denominado Trabajo, es un archivo (por convenio, se han representado los directorios por elipses y los archivos por rectángulos)
* El segundo, nombrado X, es un directorio, que a su vez tiene un hijo (que es un archivo de nombre Violeta).

Para acceder a un determinado archivo, se necesita recorrer el **camino** que conduce a él. Por ejemplo, el camino que conduce al archivo Violeta debe especificarse de la siguiente manera: /temp/X/Violeta, es decir que es un camino que comienza en el directorio raíz (/), continúa hacia el directorio temp, después sigue hasta el directorio X y finaliza en el archivo Violeta (el camino no tiene que terminar en el nombre de un archivo y pudiera haber sido /temp/X/).

Los caminos se denominan:

* Caminos absolutos. Comienzan especificando el directorio raíz (/) como primer elemento, por ejemplo: /temp/X. Obsérvese que un camino absoluto proporciona la ruta desde el directorio raíz hasta el directorio especificado en la última parte de la cadena.
* Caminos relativos. Comienzan a partir de un directorio dado que no sea el raíz, por ejemplo: X/.

Como se analizó en el capítulo II, Unix utiliza dos estructuras de datos para localizar un archivo en disco:

* La tabla de directorio
* El nodo-i (*i-node*)

Algunos nombres de directorios son estándar en todas las versiones de Unix (al menos en la mayoría), por ejemplo:

* El directorio raíz (/) es el primero que se monta cuando arranca el SO; contiene los elementos básicos para preparar el SO, también contiene puntos de montajes[47] para cualquier otro sistema de archivos que se desee montar.
* El directorio /bin aloja los comandos fundamentales.
* En el directorio /boot, residen programas y archivos de configuración que son necesarios durante el arranque del SO.
* El directorio /dev está dedicado a los manipuladores de dispositivos (*device drivers*) que controlan el acceso a los periféricos.
* El contenido del directorio /etc está conformado por programas script[48] y archivos de datos del sistema. Los archivos en el subdirectorio /etc/default contienen información que usa el sistema por defecto.
* Los directorios de inicio de los usuarios están contenidos en el directorio /home.
* Las bibliotecas para "C" y otros lenguajes de programación están en el directorio /lib.
* El directorio /root es el directorio de inicio del superusuario.
* /mnt es un directorio vacío reservado para montar sistemas de archivos.
* /tmp contiene archivos temporales creados por programas del SO.
* /usr era el directorio de inicio de los usuarios (de ahí su nombre), ahora /usr/bin contiene comandos menos comunes que no están en /bin. El directorio /usr también contiene recursos compartidos que no son críticos, por ejemplo: X Window System, Perl, etc.

Tipos de archivos

Existen tres tipos de archivos generales:

* Archivos regulares. Contienen programas y/o datos.
* Directorios. El contenido de estos archivos es una tabla de directorio.
* Archivos especiales. Corresponden a los dispositivos externos tales como terminales, impresoras, memoria, *backup*, discos, etc. Se usan para proveer un canal de acceso a los mecanismos de E/S de cada dispositivo; muchos de ellos se almacenan en el subdirectorio /dev. Sus nombres, usualmente, indican el tipo de dispositivo al que se asocian; por ejemplo, /dev/tty<n> es un manipulador de terminal, /dev/lp<n> es un manipulador de impresora, /dev/hd<n> es un manipulador de disco duro (donde n es un número entero), etc.

47 Un punto de montaje es un directorio al que se le pueden adicionar sistemas de archivos.

48 La programación en Shell script se estudia en el capítulo siguiente.

Cuando un proceso lee datos desde un dispositivo, se usa un manipulador; por ejemplo, si es /dev/tty01, los datos provienen de la terminal 01. Los archivos especiales son una interfaz entre los programas de aplicación de propósito general, que necesitan ignorar los detalles del *hardware*, y las rutinas internas del núcleo (*kernel*).

Algunos dispositivos de E/S se manipulan carácter a carácter, por ejemplo, la terminal. Los archivos especiales que proveen un enlace con los dispositivos de E/S orientados a carácter se denominan **archivos especiales de caracteres**.

Otros dispositivos de E/S transfieren grupos de datos que se denominan bloques. Esos archivos reciben el nombre de **archivos especiales de bloques**.

Muchos dispositivos de E/S de bloques también tienen una interfaz especial de caracteres denominada interfaz **raw** y comúnmente se usan por programas que ejecutan funciones de mantenimiento del SO.

Los nodos-i de los archivos especiales contienen una referencia al manejador del SO, que se ocupa de ese dispositivo concreto. Estos archivos especiales se pueden acceder igual que los archivos ordinarios, teniendo en cuenta las limitaciones de cada dispositivo.

Otra categoría son los archivos de enlace, los cuales permiten que varios nombres se asocien a un único archivo. Esta facilidad permite que se pueda tener acceso a un mismo archivo desde diferentes lugares aunque físicamente solo exista uno.

IV.3 LOS INTÉRPRETES DE COMANDOS DE UNIX

El Shell o intérprete de comandos brinda una interfaz, en modo texto, que se utiliza para que los usuarios tecleen órdenes que el Shell debe interpretar para después ordenar su ejecución. También se consideran bajo ese nombre las interfaces gráficas, aunque tradicionalmente y por razones históricas, la palabra se asocia casi siempre con la interfaz de texto muy popular en los sistemas de la familia Unix. La palabra Shell (concha) se usa con el objetivo de dar la idea de que es algo que rodea al SO y sirve de interfaz entre él y los usuarios.

En Unix, se pueden usar distintos intérpretes de comandos. A continuación, se relacionan algunos:

* El Bourne Shell (**sh**) es el más común de todos, debido a que a partir de él se han desarrollados otros.
* El Korn Shell (**ksh**) se derivó del Bourne Shell y lo desarrolló David Korn de los Laboratorios AT&T.
* El C Shell (**csh**), basado en el lenguaje C. Lo desarrolló Bill Joy en la Universidad de Berkeley, California.
* El **tcsh** es una versión del C shell con una línea de comandos interactiva. La **t** al inicio del nombre la tomó su autor, Ken Greer, del SO TENEX.

El Bourne Shell y el Bourne Again Shell

El Bourne Shell fue creado por Stephen Bourne en los Laboratorios Bell; generalmente, se puede localizar el archivo ejecutable de este intérprete de comandos en la ruta /bin/sh. La

Free Software Foundation[49] se basó en el Bourne shell e incorporó características de otras shell para desarrollar el Bourne Again Shell o **Bash** (por lo regular localizado en /bin/bash). Las ideas que se discuten en el resto del documento están relacionadas con el Bash.

Aunque se pueden encontrar varios libros que describen el Bash, es importante tener a mano la documentación que viene con el manual de ayuda de la versión de Unix que se esté utilizando, para lo cual se usa el comando man bash.

Durante el proceso de entrada al sistema (*logging*), Unix resuelve varias tareas para preparar el sistema, lo que incluye el arranque del intérprete de comando. Como parte de su trabajo inicial, el Shell busca en lugares especiales para obtener información que necesita; uno de esos lugares es el directorio /home del usuario que está entrando al sistema. Los archivos buscados pueden variar de un Shell a otro pero la idea central es analizar algunos archivos especiales y resolver lo que en ellos se indica. Una de las acciones que se lleva a cabo es asignarles valores a las variables de ambiente (*environment*) de acuerdo con lo que se especifique en esos archivos; esa acción recibe el nombre de inicialización de las variables de ambiente.

Uso de los archivos de arranque

La idea de tener un archivo que se lee en el proceso de arranque no es privada del Shell, muchos programas hacen eso para configurarse a sí mismos y ajustarse a diferentes fines o gustos. En Unix, ese tipo de archivo recibe el nombre genérico de archivo rc (archivo de recursos) algunos de los más comunes son .exrc (lo usan los editores vi y ex), .mailrc (lo usan varias herramientas de correo electrónico), .cshrc (lo usa el C-Shell), etc. Los archivos rc normalmente se encuentran en el directorio /home de cada usuario.

Estos archivos se llaman **archivos punto** debido a que sus nombres comienzan con un punto. Los archivos punto son ocultos.

IV.3.1 Trabajo con el Shell

Cada vez que se teclea un comando desde una terminal, se está interactuando con un Shell, el cual se encarga de interpretar los comandos y ordenar su ejecución. Se pueden enviar a ejecutar lotes de trabajo, comandos de superficie y comandos de fondo.

El Shell también trabaja con variables que se usan con diferentes fines, esas variables pueden ser propias de él o pueden ser variables de usuarios. En general, existen cuatro tipos de variables:

* De ambiente o entorno.
* De usuarios.
* Especiales o predefinidas.
* Posicionales o parámetros.

El nombre de una variable es una cadena alfanumérica que no puede comenzar con dígito. Ellas toman valor cuando aparecen a la izquierda de un signo de asignación (el signo =) y puede quitársele el valor con el comando unset.

49 http://www.fsf.org/

Variables de ambiente (*environment*) o de entorno

Las variables de ambiente contienen valores que pueden afectar el comportamiento de ciertos procesos y también se usan en otros SO y en diversos entornos de programación.

A continuación, se muestra una lista de algunas variables de entorno de los SO tipo Unix (sus nombres se escriben en mayúscula):

* BASH. Contiene el camino donde está el intérprete de comandos Bash, por ejemplo: /bin/bash.
* HISTFILE. Contiene el camino donde está un archivo con la historia de los comandos tecleados, por ejemplo: /home/mlezcano/.bash_history.
* HOME. Contiene el camino al directorio /home del usuario, por ejemplo: /home/mlezcano.
* HOSTNAME. Contiene el nombre de la computadora, por ejemplo: ubuntu-01.
* OSTYPE. Contiene el nombre del SO que está instalado, por ejemplo: Linux-GNU.
* PATH. Contiene un conjunto de caminos, separados por el signo dos puntos (:). Cuando se ordena ejecutar un comando, el Shell busca en los directorios especificados por esos caminos, y si el comando no está en ninguno de ellos, el usuario recibe un mensaje de error diciéndole que el programa no existe. Un ejemplo del contenido de la variable PATH es el siguiente:

  /usr/local/sbin:/usr/local/bin:/usr/bin:/sbin

* PS1. Contiene una secuencia de órdenes de formato para imprimir el *prompt*[50] del Shell; por ejemplo, la asignación PS1="\u@\h\w>" hará que el prompt esté formado por: el nombre del usuario (u - username), el nombre del host (h - hostname), el camino hasta el directorio de trabajo (w - working dir) y un signo 'mayor que' (>).
* SHELL. Define el camino donde está el Shell que usa el usuario, por ejemplo: /bin/bash.
* TERM. Especifica el tipo de terminal que está usando el usuario, por ejemplo: xterm.
* UID. Contiene el número de identificación del usuario (*user identification*), por ejemplo: 1011.

Son muchas más las variables de entorno, algunas de ellas son particulares del Shell que se usa y otras son de algún programa en específico. Para ver un listado completo de las variables de entorno en un ambiente Unix, se usa el comando env.

Variables de usuarios

A este tipo de variables lo define el usuario y mayoritariamente se usa cuando se programa en Shell script (es el tema del capítulo V).

50 El *prompt* es el carácter o el conjunto de caracteres que muestra el Shell para indicar que está a la espera de órdenes.

Variables especiales o predefinidas

Las variables especiales reciben valores cuando se realizan determinadas acciones y por eso no se les puede asignar valores directamente. Se usan extensivamente dentro de los programas Shell script y se representan por diferentes símbolos. A continuación, se muestran algunos ejemplos:

- # Contiene la cantidad de parámetros pasados a un programa Shell script.
- Contiene todos los parámetros pasados a un programa Shell script.
- ? Contiene el código de error del último comando que se ha ejecutado.
- $ Contiene el identificador del proceso (su PID).
- ! Contiene el identificador del último proceso ejecutado.

Variables posicionales o parámetros

Las variables posicionales contienen los parámetros pasados a un programa Shell script. Estas variables tienen nombres numéricos que reflejan el orden en que se pasa el parámetro: 0, 1, 2, 3,…, n. El número 0 es el nombre del programa; si se imprime el contenido de esa variable fuera de un programa, la cadena obtenida será el nombre del Shell.

Para referenciar el contenido de una variable cualquiera, se debe preceder del signo $; por ejemplo, la orden a=4 hace que se le asigne el valor 4 a la variable de usuario a, si se deseara sumarle dos a esa variable (en un entorno bash), se tendría que hacer de la forma siguiente: a=$(($a+2))[51]. En este caso, $() es la orden para evaluar una expresión y la expresión a evaluar es ($a+2), es decir, sumarle 2 al contenido de la variable a.

Los valores de las variables posicionales también se pueden fijar usando el comando set; por ejemplo, realice el ejercicio siguiente:

1. Ejecute el comando who, su salida puede ser algo como la siguiente:

   dlezcano tty7 2015-09-29 10:36

   Esta acción solo se hace para que verifique la salida del comando who.

2. Ejecute ahora el comando set `who`

   Esta orden hará que se ejecute el comando who (porque está entre apóstrofos invertidos) y el comando set tomará los valores de salida de who (observe el paso 1) **para fijar las variables posicionales**.

3. Ejecute ahora el comando echo $0 y observe que se imprime el nombre del Shell, por ejemplo: bash; este valor no se obtuvo porque se haya ejecutado set `who`, sino porque el programa que está ejecutando la orden echo $0 es el Shell, en particular el bash.

51 La construcción con doble paréntesis permite manejar las variables al estilo C, por ejemplo: ((a=10)); ((a++)).

4. Ejecute el comando echo $1 y observe que se imprime el nombre del usuario que está conectado: dlezcano en el ejemplo de acuerdo con el paso 1.
5. Ejecute el comando echo $2 y observe que se imprime el identificador de la terminal desde la que está conectado el usuario dlezcano: tty7, de acuerdo con el paso 1.
6. Ejecute echo $3 y observe que se imprime la fecha en que fue ejecutado el comando: 2016-09-29.
7. Ejecute echo $4 y observe que se imprime la hora en que fue ejecutado el comando: 10:36.

Otros ejemplos:

- ✓ echo $PATH. Imprime el contenido de la variable de ambiente PATH.
- ✓ echo $m. Imprime el contenido de la variable de usuario m.
- ✓ echo $$. Imprime el contenido de la variable especial $.

IV.3.2 Procesos manipulados desde la terminal

En el libro, se ha hablado bastante acerca del concepto de proceso, pero es bueno insistir en él. Un proceso es un programa en ejecución, y en este sentido, es mucho más que un programa dado que es un **ente activo** que tiene recursos asignados; uno de los más importantes es el procesador.

El núcleo o *kernel* del SO provee el control y garantiza que varios procesos puedan usar la CPU en diferentes instantes de tiempo. Un proceso ejecuta por corto tiempo y entonces el control se pasa a otro proceso, así se logra que la CPU sea un recurso compartido (debe verse el capítulo I).

Dentro del SO Unix existen prioridades para obtener el procesador, solo el superusuario puede alterar esas prioridades. Los procesos "nacen" cuando comienzan a ejecutarse y "mueren" un poco después de terminar de ejecutar su última instrucción, ellos pueden actuar como procesos de superficie o procesos de fondo.

Desde el mismo arranque del SO y antes de que el usuario teclee cualquier orden ya existen varios procesos actuando en el sistema. Esos procesos son esenciales para el trabajo del SO y solo mueren cuando el SO termina o si se eliminan explícitamente, si es que está permitido hacerlo (algunos no se pueden eliminar debido a que son fundamentales para el trabajo del SO).

Existe una jerarquía de procesos en Unix. El comando pstree (figura IV.4) permite apreciarla parcialmente:

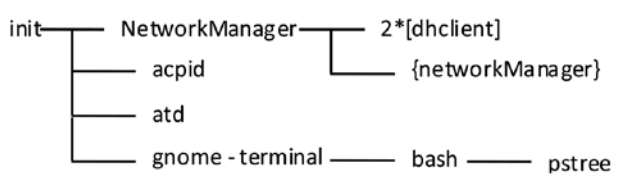


Figura IV.4. Vista parcial del árbol de procesos.

El árbol de procesos que se obtiene con pstree es mucho mayor que el mostrado en la figura IV.4, aunque la orden se ejecute como primer comando después de iniciar el SO.

La posición cimera de la jerarquía de procesos la ocupa el proceso init (*initialization*), que es el primero que se ejecuta después de que se carga el núcleo del SO. Este proceso se ejecuta como un demonio[52] y tiene el identificador 1. De init se deriva toda la jerarquía de procesos.

Algunos ejemplos de procesos que actúan como demonios son:

* vhand, es un proceso permanente del SO.

* lpsched, que se deriva del subsistema de impresión lp.

El inicio del proceso init marca el fin del procedimiento de arranque del SO; init deberá realizar diversas tareas que dependerán de cada implementación, pero en general tendrá que verificar el sistema de archivo, iniciar algunos servicios y las terminales. El proceso init nunca se puede eliminar.

Los procesos que se van creando son hijos de algún proceso (excepto init). Cuando un proceso crea a otro, se establece una relación padre-hijo que permanece hasta que el hijo o el padre terminen. En el caso de que el padre termine primero que el hijo, se dice que el proceso hijo queda **huérfano** y el proceso init asume la paternidad de ese hijo huérfano. Observe que en la figura IV.4 uno de los procesos es el bash, es decir, el intérprete de comandos o Shell, el cual tiene a pstree como hijo. Todos los procesos que se ordenan ejecutar desde la terminal son hijos del Shell.

Cuando un usuario se conecta a un sistema Unix, el SO crea un proceso Shell que es la interfaz entre el usuario y el SO, ese proceso desaparece o muere cuando el usuario se desconecta. Al invocar un comando o programa desde el Shell (ya sea un comando del SO o un programa de aplicación creado por cualquiera), el programa invocado pasa una transición que lo convierte en un ente activo llamado proceso. Los procesos que se ejecutan desde una terminal pueden ejecutarse como procesos de superficie o de fondo.

Los procesos de superficie (*foreground*) se caracterizan por tomar datos de la entrada estándar (el teclado) y escribir hacia la salida estándar (el monitor), es decir que interactúan con el usuario. Cuando el Shell ejecuta un proceso de superficie, se queda en espera de él y por eso no es posible ordenar la ejecución de otro proceso desde esa misma terminal (el Shell no devolverá el *prompt* hasta que su hijo termine).

Los procesos también se pueden ejecutar como procesos de fondo (*background*), el Shell no espera a que los procesos de fondo terminen y por eso devuelve el *prompt* después de que el proceso comienza su ejecución, con lo cual permite ordenar la ejecución de nuevos programas. El operador & (*ampersand*), situado al final de un comando, es la orden para que el Shell ejecute procesos de fondo.

Un proceso que se ejecuta como proceso de fondo resultará eliminado al terminar la sesión de trabajo, aun cuando el proceso en cuestión no haya finalizado; esto se debe a que el comando es hijo del Shell que se cargó al iniciarse la sesión de trabajo.

52 Demonio (daemon - *disk and execution monitor*). Proceso que se ejecuta en segundo plano, no interactúa con el usuario; realiza operaciones específicas en tiempos predefinidos o en respuesta a ciertos eventos.

Ejemplos de comando ejecutando como proceso de superficie y de fondo

El comando ls / -l -R y su forma equivalente ls / -lR (la segunda forma es más compacta), usan las opciones -l (*long format*) y -R (*recursive*). Esta orden hace que se haga un recorrido (en forma recursiva) por todos los hijos del directorio especificado que, en este caso, es el directorio raíz o *root* (/). Como el directorio raíz es el ancestro más exterior de todos los directorios, el listado que se obtendrá se demorará bastante y durante ese tiempo no se podrá hacer nada desde la terminal, porque el Shell estará esperando por su hijo y no devolverá el *prompt* hasta que termine.

El comando anterior se ejecuta como un proceso de superficie. Si se le agrega el símbolo *ampersand* (&) al final (ls / -lR&), se ejecutará como un proceso de fondo y, por tanto, el Shell debería devolver el *prompt* inmediatamente después de iniciar la ejecución del proceso; sin embargo, no sucede así y habrá que esperar a que el proceso termine para ejecutar otro comando. ¿Por qué sucede eso? Ahora la razón es que la terminal estará ocupada por la salida del comando ls, que se está ejecutando de fondo pero envía su salida a la terminal que comparten padre e hijo (el Shell y el comando).

Para resolver el problema, se debe redirigir la **salida estándar** hacia otro lugar que no sea la terminal. En Unix esa redirección se logra con el símbolo >. También se puede redirigir la **entrada estándar** con el símbolo < (no es necesario en este ejemplo). La nueva versión del comando queda en la forma siguiente: ls -lR > directorio&.

El comando anterior hace que la salida del comando ls, que debería ir hacia el equipo de salida estándar (la pantalla de la terminal), se redirija al lugar especificado después del símbolo >, es decir, un archivo llamado directorio. Ese archivo se creará en ese instante, pero si existe será sobrescrito.

En este caso, la salida va a demorar menos, pero a no ser que la persona que ordena el listado sea el superusuario, se seguirá viendo un listado en la terminal. El listado que se apreciará es de los errores que se producirán al tratar de entrar en directorios sobre los cuales no se tiene permiso (los mensajes de error salen también por la terminal).

Para solucionar este problema, se debe redireccionar la salida de **error estándar** también. El manipulador (*handle*) que tiene asociada la salida estándar en Unix es el número 2 (la entrada estándar es el 0 y la salida estándar es el 1). El nuevo comando se muestra a continuación:

```
ls -lR / >directorio 2>error&
```

Una vez recibida esta orden, el SO devuelve algo como lo siguiente:

```
[1] 18419
$
```

El número entre corchetes, 1, es el identificador del conjunto de trabajo y el otro número, 18419, es el identificador del último proceso del grupo. Obsérvese que el Shell devuelve el *prompt* de inmediato y por eso se podrán ejecutar otros comandos mientras el comando ls -lR / >directorio 2 > error& se ejecuta como comando de fondo. El listado deseado quedará en el archivo directorio y los errores, si los hay, quedarán en el archivo error.

Envío de procesos al fondo y a la superficie

Una vez iniciado un proceso como proceso de fondo o de superficie, se puede enviar a la superficie o al fondo, respectivamente, para lo cual se usan los comandos fg y bg.

Cuando se inicia un proceso (ya sea de superficie o de fondo), se puede suspender el proceso temporalmente para después reanudarlo en la forma que se desea. Para poder entender bien el concepto, se hará en forma práctica.

Se invocará, desde la terminal, un proceso que no se detiene. Este es el caso de xload, que muestra periódicamente un histograma actualizado del promedio de carga del sistema. Se puede proceder de la siguiente manera:

1. Desde una terminal se invoca el comando xload, al cual no se le agregará el símbolo & al final. En ese caso, se ejecutará como un proceso de superficie y el Shell no devuelve el *prompt*.

   Después de la acción anterior, no se puede hacer nada en la terminal porque el Shell no ha devuelto el *prompt* y está en espera de su proceso hijo (en este caso xload).

2. Sin embargo, sí se puede suspender el proceso actual (xload), para lo cual se usa la combinación de teclas <ctrl><z> (las teclas control y z a la vez). Al hacer esa acción, se verá algo como lo siguiente: [1]+ Stopped

   Lo que indica que el trabajo 1, (job 1)[53], se ha detenido.

   Ahora es posible enviar ese trabajo de nuevo a la superficie en la forma siguiente: fg %1 o al fondo: bg %1

Trabajo con procesos

Los procesos se pueden comunicar entre sí a través de señales. Entre otras, existen señales para:

* Matar procesos.
* Referenciar varias condiciones de errores.
* Comunicarle a la terminal que la máquina ha sido apagada.

La ejecución de los procesos está dividida en dos niveles: usuario y *kernel*. Los procesos en modo usuario pueden acceder a sus propias instrucciones y datos, pero no a instrucciones y datos del *kernel* (o de otros procesos). Los procesos en modo *kernel* pueden acceder al núcleo del SO y a direcciones de usuarios. Las instrucciones que solo se pueden ejecutar en modo *kernel* se denominan **instrucciones privilegiadas**.

Una señal es una notificación asíncrona que se envía a un proceso (pesado o ligero) para notificarle la ocurrencia de un evento. Cuando se envía una señal, el SO interrumpe el proceso destinatario para entregársela; el proceso que reciba la señal la tratará de una forma particular si tiene definido un manipulador para tratarla, y en caso de no tener ese manipulador, se le da el tratamiento por defecto. La tabla IV.1 muestra algunas señales.

53 Un *job* o trabajo es un conjunto de procesos. El conjunto puede estar formado por un solo proceso.

Tabla IV.1.

Señales en sistemas POSIX

NOMBRE SIMBÓLICO	NÚMERO	USO O EFECTO
SIGHUP (*hang-up*)	1	Reportar que una terminal de usuario está desconectada (puede ser por rotura en la red o la línea telefónica).
SIGINT (*program interrupt*)	2	Se envía a un proceso cuando el usuario teclea la señal de interrupción (ctrl c).
SIGQUIT	3	Es similar a SIGINT, pero la provocan caracteres diferentes.
SIGILL (*illegal instruction*)	4	Usualmente significa que el proceso está tratando de ejecutar instrucciones no permitidas, típicamente SIGILL indica que el archivo ejecutable está dañado o que se ha intentado ejecutar datos. También puede generarse por *overflow* en la pila (*stack*) o cuando el sistema tiene problemas ejecutando un manipulador de señales.
SIGTRAP (*tramp*)	5	Lo genera una instrucción de máquina de punto de ruptura u otra instrucción de trampa. Los *debuggers* usan esta instrucción.
SIGABRT (*abort*)	6	Indica un error detectado por el proceso en sí, que lo reporta llamando a *abort*.
SIGFPE	8	Error aritmético fatal, por ejemplo: división por cero, *overflow*, etc.
SIGKILL (*kill*)	9	Provoca la inmediata terminación del proceso. No puede manipularse, ignorarse ni bloquearse.
SIGBUS	10	Se genera cuando se derreferencia un apuntador no válido (típicamente uno no inicializado). Mientras SIGSEGV indica un acceso no válido a una dirección valida de memoria, SIGBUS indica un acceso a una dirección no válida.
SIGSEGV (*segmentation violation*)	11	Se genera cuando un proceso intenta leer o escribir fuera de la memoria que tiene asignada, o cuando trata de escribir en una memoria de solo lectura.
SIGSYS	12	Trampa al SO con un número no válido.
SIGPIPE (*broken pipe*)	13	Se envía a un proceso que trata de escribir hacia un *socket* sobre el cual no se puede escribir o no está conectado.
SIGALRM	14	Indica que el intervalo de tiempo (real) especificado en una llamada a alarm o alarmd ha expirado.
SIGTERM	15	Es una señal genérica usada para provocar la terminación de un proceso. Contrario a SIGKILL, la señal se puede bloquear, manipular e ignorar. El comando kill, del Shell, genera una señal SIGTERM por defecto.
SIGUSR1 (*user signal 1*)	16	Señal para que los usuarios usen.
SIGUSR2 (*user signal 2*)	17	Señal para que los usuarios usen.
SIGCHLD (*Child Status*)	18	Se envía al padre de un proceso cuando su hijo termina, es interrumpido, o se reinicia después de haber sido interrumpido.
SIGPWR (*Power Fail/ Restart*)	19	Señal de fallo de energía.

SIGWINCH (*Window Size Change*)	20	La terminal controlada por el proceso ha cambiado de tamaño.
SIGURG (*Urgent Socket Condition*)	21	Se envía cuando se recibe un dato *out-of-band - OOB* sobre un *socket* que soporta ese tipo de dato.
SIGPOLL (*Socket I/O Possible*)	22	Informa que existe una entrada/salida disponible.
SIGSTOP (*stopped*)	23	Señal de parada, debe usarse en conjunción SIGCONT.
SIGTSTP (*terminal stop*)	24	Se envía cuando el usuario teclea ctrl z en una terminal.
SIGCONT (*continued*)	25	Se debe enviar a un proceso que se ha detenido con SIGSTOP para que pueda continuar.
SIGTTIN (*tty input*)	26	Señal que envía el SO a los procesos de fondo cuando tratan de leer desde la terminal. La respuesta típica es detener el proceso hasta que llegue una señal SIGCONT, lo cual sucederá cuando el proceso pase a la superficie.
SIGTTOU (*tty output*)	27	Es similar a SIGTTIN, pero en este caso el proceso de fondo intenta escribir en la terminal.
SIGVTALRM (Virtual Timer Expired)	28	Es muy parecida a SIGALRM, pero al contrario de ella no mide tiempo real de ejecución sino un cierto tiempo que lleva el proceso corriendo.
SIGPROF (Profiling Timer Expired)	29	Muy parecida a SIGALRM (tiempo real) y a SIGVTALRM (tiempo virtual), pero ocurre después de que el proceso ha pasado un tiempo ejecutando su propio código más otro tiempo que el SO ha dedicado a ejecutar código a nombre del proceso.
SIGXCPU (*CPU time limit exceeded*)	30	La envía el SO a un proceso que ha excedido su límite de tiempo de uso de la CPU.
SIGXFSZ (*file size limit exceeded*)	31	El SO la envía a un proceso que intenta crear un archivo de una longitud mayor al límite permitido.
SIGWAITING (*all LWPs blocked*)	32	Cuando todos los hilos de un proceso están bloqueados en una espera indefinida, el SO envía esta señal al proceso.

Las señales son constantes simbólicas, sus nombres comienzan siempre por SIG (*signal*) y se escriben en mayúscula. Aunque las señales se pueden referir por su nombre simbólico o por su número, es mejor hacerlo en la primera forma para evitar diferencias de una implementación a otra.

Pasos que da el SO para ejecutar un proceso desde el Shell

Cuando se ordena la ejecución de un programa, se sigue un conjunto de pasos que se describen a continuación.

1. El Shell lee la línea que contiene el comando, puede incluir argumentos u opciones.
2. Verifica, sintáctica y semánticamente, la línea leída.

3. El Shell crea un hijo dividiéndose en dos copias idénticas (ocurre una bifurcación). Cada copia tiene los mismos archivos abiertos y el mismo código de programa, pero cada parte puede averiguar si él es el original o es la copia.
    - El proceso original, es decir el **padre**, es el Shell.
    - El nuevo proceso, el comando, es su **hijo**.
4. El Shell usa la llamada al sistema wait para informarle al núcleo del SO que esperará a su hijo.
5. El proceso hijo ejecuta la llamada al sistema execute con los parámetros leídos en el paso 1.
6. El núcleo responde a la llamada al sistema execute (se cambia a modo protegido) cargando el nuevo programa, es decir que sustituye el código heredado por el código del comando (solo cambia el código y todos los archivos abiertos continúan disponibles).
7. El núcleo cambia a modo usuario y le da el control al proceso hijo para que ejecute el comando.
8. Cuando el proceso hijo finaliza, envía una llamada de terminación al sistema (exit) y el SO reacciona pasándose a modo protegido para finalizar el proceso hijo. Al ejecutar exit, se cierran todos los archivos del proceso, que quedará en un estado llamado zombie[54] hasta que su padre pregunte por él.
9. La muerte del hijo despierta al padre, es decir, al Shell.
10. El padre recibe el estado de retorno de su hijo y se borra lo que queda del proceso zombie.
11. En ese momento, el Shell devuelve el *prompt* y queda en espera del siguiente comando.

IV.3.4 Comandos para el trabajo con procesos

Existen muchos comandos en un SO tipo Unix. Algunos son particulares de determinada distribución y no es objetivo de este libro ofrecer información acerca de todos esos comandos; sin embargo, resulta muy útil tener una idea general de algunos que se relacionan con los procesos, dado que su uso puede facilitar muchas tareas, lo que incluye las relacionadas con el entorno de programación que brinda el SO. En el resto de este apartado, se presenta una selección de esos comandos.

Comando ps

El comando ps hace un reporte de los procesos que están en ejecución. El comando tiene varias opciones que pueden diferir de una implementación a otra. A continuación,

54 Un proceso queda en estado zombie cuando muere primero que su padre. En ese caso, el *kernel* vacía el espacio de direcciones del hijo pero retiene el proceso en su tabla de procesos (en el estado zombie). Un proceso en este estado puede eliminarse porque no es realmente un proceso.

se muestran algunas de ellas (debe usarse la ayuda para ver las que están disponibles en cada caso: man ps):

- -A, -e. Son opciones equivalentes y muestran información de todos los procesos.
- -T. Muestra información de todos los procesos que se han iniciado desde la terminal actual.
- -U <usuario>. Presenta información de todos los procesos que se ejecutan a nombre del usuario especificado, por ejemplo: ps -U mlezcano.
  A continuación, se puede ver una salida típica del comando:

```
PID        tty      time         cmd
14662      pts/0    00:00:01     bash
15030      pts/0    00:00:00     ps
```

La salida anterior muestra: en la primera columna, el PID (*process identification*) del proceso; en la segunda columna, la terminal desde donde se inició el proceso; en la tercera columna, el tiempo de ejecución, y en la última, el comando (CMD) que se invocó. Obsérvese que bash y ps son programas, mientras que 14662 y 15030 son los procesos que nacieron a partir de la ejecución de esos programas.

Si un usuario ordena la ejecución del mismo programa varias veces, se repetirán los nombres de la última columna cuando se ordene ejecutar ps, pero no se repetirán los números (PID) de la primera columna; es decir que si ordena ejecutar un programa P, por ejemplo, tres veces, el nombre P estará tres veces en la última columna, pero cada una de esas órdenes habrá dado vida a procesos diferentes (sus PID así lo indicarán), que son instancias de un mismo programa. Aquí se puede apreciar de nuevo la diferencia sustancial que existe entre proceso (programa en ejecución, con vida propia) y programa (simple listado de órdenes, sin vida).

```
$xload&
[1] 2554
$xload&
[1] 2556
$ps
PID       TTY      TIME         CMD
1362      pts/0    00:00:00     bash
2554      pts/0    00:00:00     xload
2556      pts/0    00:00:00     xload
2557      pts/0    00:00:00     ps
$jobs
jobs
[1]-  Running
[2]+ Running
$
```

Figura IV.5. Terminal mostrando el resultado de teclear varios comandos.

La figura IV.5 simula una terminal que muestra el resultado de algunas órdenes. En adelante, se usará este tipo de figura que adopta el siguiente convenio: los mensajes del SO se mostrarán en negrita y lo que teclee el usuario aparecerá sin resaltar.

Las órdenes xload& fueron dadas dos veces para que las dos instancias de xload se ejecutaran como proceso de fondo, debido a que si se hubiera hecho como proceso de superficie en la primera orden, no devolvería el *prompt*, lo cual impediría escribir la segunda orden; y si se hiciera como proceso de superficie en la segunda orden, tampoco devolvería el *prompt* y no permitiría más órdenes (todo esto por las características particulares del programa xload).

Comando Jobs

El comando jobs muestra el estado de los trabajos que se están ejecutando desde la terminal actual. La salida del comando en la forma vista en la figura IV.5 tiene el siguiente significado:

La palabra *running* informa que los trabajos 1 y 2 ([1], [2]) están activos, es decir que no han sido interrumpidos por ninguna señal, ni han terminado.

La combinación de los símbolos [2]+ indica que el trabajo 2 se usará por defecto por los comandos fg y bg, a fin de enviar el proceso a la superficie o al fondo, respectivamente.

La combinación de los símbolos [1]- indica que el trabajo 1 pasará a ser usado por defecto por los comandos fg y bg cuando el trabajo 2 termine.

Además de *running*, pueden existir las siguientes cadenas para especificar el estado de los trabajos (*jobs*):

* Done. Indica que el trabajo completó su tarea con éxito (código de retorno 0).
* Done(code). Indica que el trabajo terminó su tarea con éxito, pero con un código de retorno (code) distinto de cero.
* Stopped o Stopped(signal). La palabra Stopped también puede ser Suspended según la implementación. Indica que el trabajo fue suspendido por una señal SIGTSTP o por la señal (*signal*) especificada: SIGTSTP, SIGSTOP, SIGTTIN, SIGTTOU.

Comando kill

El comando kill envía una señal a un proceso. La señal por defecto es la de terminación (TERM). El proceso que recibe una señal de terminación detiene su ejecución abruptamente. Con la opción -l se pueden ver todas las señales disponibles (kill -l).

La figura IV.6 muestra el uso del comando kill suponiendo que después de las órdenes dadas en la figura IV.5 no se ha hecho ninguna otra acción que afecte los trabajos (*jobs*) involucrados.

Obsérvese que después de dar la orden de terminar al trabajo 1, se ha usado nuevamente el comando jobs para ver el estado de los trabajos y se puede apreciar que dicho trabajo ha terminado.

```
$kill %1
$jobs
[1]-  Terminated xload
[2]+ Running xload&
$
```

Figura IV.6. Eliminación de un trabajo (*job*) desde una terminal.

Los procesos individuales se pueden eliminar enviándole una señal de terminación en la forma: kill <PID>; como la señal por defecto es TERM, el proceso identificado por el PID terminará; por ejemplo, kill 2587 terminará (o eliminará) el proceso 2587, y después de la orden anterior aparecerá la cadena *Terminated* en la terminal desde donde se ejecutaba el proceso 2587.

```
$ls -lR / &
drwxr-xr-x 2 2 root 4096 2011-09-24 23:53 bin
drwxr-xr-x 2 2 root 4096 2016-10-16 07:23 boot
…
…
$[1] killed ls - - color = auto –lR /
$tty
/dev/pts/0
$
```

```
$ps -U mlezcano
PID      TTY      TIME CMD
2814     pts/1    00:00:00    bash
2846     pts/0    00:00:00    ls
2847     pts/1    00:00:00    ps
$kill -9 2846
$tty
/dev/pts/1
$
```

Figura IV.7. Eliminación de un proceso desde una terminal: izquierda terminal 1, derecha terminal 0.

Otra forma de uso del comando kill se muestra en la figura IV.7, en la cual también se eliminan procesos y no trabajos. Se parte del ejemplo ls -lR / &, que se analizó antes (parte izquierda de la figura); como ya se vio, esa acción provoca un largo listado del directorio que quizás se desee interrumpir (se ha representado por puntos suspensivos). ¿Qué hacer si no se puede hacer nada porque la terminal está ocupada y el comando de fondo no se puede interrumpir? En esos casos, se procede de la siguiente forma:

1. Se abre una nueva terminal (es la que se muestra a la derecha).
2. Desde la nueva terminal se ejecutan las órdenes mostradas en la figura IV.7 a la derecha.
3. El comando ps -U <username> (en este caso, mlezcano es el <username>) muestra un listado con el PID de todos los procesos que ese usuario ha ordenado ejecutar, desde cualquier terminal, incluye la que está ocupada por la salida (a la izquierda de la figura) y la que se acaba de abrir.
4. Obsérvese que el comando ps se ha ordenado desde la nueva terminal que se ha abierto, pts/1 (derecha en la figura IV.7), y que el listado muestra que el usuario mlezcano ordenó un comando ls desde la terminal pts/0 (izquierda en la figura).

5. El proceso ls tiene el PID 2846 y es el que se desea eliminar, lo cual se hace enviándole una señal de terminación con el comando kill -9 2846, lo que provoca que se detenga.
6. Observe el resultado de detener el comando en la terminal pts/0, donde el SO devuelve la cadena [1] killed ls - - color = auto –lR /, con lo que hace notar que el proceso que tenía ocupada la pantalla ha sido eliminado. En este caso, se ha usado el número 9 en lugar de la opción TERM para especificar la señal que se enviará al proceso, dado que son equivalentes; en todo caso, no era necesario ninguno de los dos porque esa es la señal por defecto.
7. Finalmente se usa el comando tty para mostrar los nombres de ambas terminales.

Comando wait

El comando wait espera a que un proceso termine. Si no se le especifican argumentos, espera por todos los procesos hijos del proceso que se ejecuta en *background*. Si el proceso especificado termina anormalmente, debido a que recibe una señal, se puede analizar el valor de retorno que está contenido en la variable especial ?, debido a que los procesos que terminan normalmente dejan en esa variable el valor cero y los que terminan anormalmente deben dejar un valor diferente de cero.

Comando nohup

El comando nohup ejecuta el comando asociado, command, ignorando las señales de terminación y debe teclearse: nohup <command>& para que el comando asociado se ejecute como tarea de fondo. Si el comando hace su salida hacia la terminal, su salida y la salida de error quedarán en el archivo nohup.out; si no es posible ponerla en ese lugar, la pondrá en $HOME/nohup.out; si no puede hacerlo en ninguno de esos dos lugares, no se ejecuta el comando.

Comando nice

Permite cambiar la prioridad a un proceso. Las prioridades son números enteros con signos, por ejemplo, desde -20 (la mayor prioridad) hasta 19 (la menor prioridad).

Comando at

Ejecuta un comando en un instante dado. Se puede usar para iniciar tareas en horas de la madrugada, con el objetivo de hacer ciertos chequeos que consumen tiempo y recursos.

Comando top

El comando top es una herramienta poderosa y configurable que proporciona una vista dinámica, en tiempo real, del SO. La información se renueva (por defecto) cada tres segundos. A continuación, se muestra una salida típica del comando (solo una parte de ella):

```
top - 10:24:10 up 10 min, 2 users, load average: 0.05, 0.08, 0.07
Tasks: 133 total, 1 running, 132 sleeping, 0 stopped, 0 zombie
Cpu(s):      0.7%us,          1.0%sy, 0.0%ni, 97.7%id, 0.0%wa, 0.0%hi, 0.0%si, 0.0%st
Mem:   509264k  total,  278636k used,   230620k free,     38924k buffers
Swap:  407544k  total,       0k used,   407544k free,    118044k cached
PID   USER      PR  NI  VIRT  RES    SHR  S  %CPU  %MEM   TIME+  COMMAND
906   root      20   0  155m   18m   7636  S   0.7   3.7   0:08.76 Xorg
1283  mlezcano  20   0  44884  12m   9880  S   1.0   2.5   0:04.0  gnome-terminal
```

En la parte superior de la información proporcionada por top, se ofrecen estadísticas del sistema, tales como la carga del sistema y la cantidad total de tareas, entre otras. Los datos que aparecen entre paréntesis en las líneas siguientes hacen referencia al ejemplo mostrado:

* Primera línea: la hora actual (10:24:10); tiempo que lleva el SO activo (10 min); cantidad de usuarios utilizando el sistema (2); la carga del sistema durante: el último minuto (0.05), los últimos cinco minutos (0.08) y los últimos 15 minutos (0.07).
* Segunda línea o Tasks: cantidad total de procesos (133) y por estados. Los estados pueden ser:
  - *running* (1): procesos en estado de ejecución (tienen la CPU, uno solo en cada procesador) más los que están en estado de listos.
  - *sleeping* (132): procesos dormidos esperando que ocurra un evento para volver al estado *running*.
  - *stopped*: procesos detenidos. Generalmente, un proceso entra en este estado cuando recibe alguna señal que tiene como acción por defecto pasar el proceso al estado *stopped*. El proceso no se detendrá si tiene un manejador de señal que especifique otra acción ante esa señal. La señal SIGCONT saca al proceso de este estado.
  - *zombie*. Un proceso queda *zombie* cuando termina antes que su padre.
* Tercera línea o CPU(s): muestra los porcentajes de uso del procesador, clasificados de acuerdo con lo que se hace:
  - us: tiempo de procesos de usuario.
  - sy: tiempo de procesos del sistema.
  - ni: tiempo de procesos del usuario con nice positivo.
  - wa: tiempo que pasan los procesos esperando por pedidos de entrada/salida.
  - id: tiempo inactivo debido a que ningún proceso necesita el procesador, ni hay pedidos de entrada/salida pendientes.
  - st: *steal time*. Se aplica cuando el SO se ejecuta sobre una máquina virtual. Este tiempo de espera es cuando el hipervisor no le asigna tiempo a la máquina virtual.

* Cuarta línea o Mem: uso de la memoria, detallada como: cantidad total de memoria (509264k), memoria utilizada (278636k), memoria libre (230620k), memoria usada como búfer (38924k).
* Quinta línea o Swap: Uso de la memoria virtual.

Después del resumen, aparecen los datos de los procesos activos, ordenados según el uso del procesador; el orden puede cambiar constantemente debido a que esta información se proporciona en tiempo real. El significado de cada columna se detalla a continuación:

PID: identificador del proceso.

USER: usuario que inició el proceso.

PR: prioridad del proceso.

NI: valor *nice*.

VIRT: cantidad de memoria virtual utilizada.

RES: cantidad de memoria física utilizada.

SHR: cantidad de memoria compartida

S: (*status* - estados) del proceso: D (*uninterruptible sleep*), R (*running*), S (*sleeping*), T (*traced stopped*), Z (*zombie*).

%CPU: porcentaje de CPU utilizado desde la última actualización de la pantalla.

%MEM: porcentaje de memoria física utilizado.

TIME+: Tiempo total de CPU que el proceso ha utilizado desde que se inició.

COMMAND: comando que inició el proceso.

Existe una versión más amistosa de top denominada htop, que puede instalarse adicionalmente.

Diagnóstico de problemas con los procesos

Una manera de percatarse de alguna anormalidad dentro del sistema es examinar la lista de procesos. Por ejemplo, las siguientes situaciones pueden sugerir que hay algún problema:

* La existencia de muchos procesos activos (la palabra "muchos" dependerá de la capacidad del SO y se necesita práctica para conocer esto).
* Un proceso acumula demasiado tiempo de CPU.
* Procesos con demasiados hijos.
* El tiempo de respuesta de la máquina se vuelve repentinamente muy lento.

Todas estas situaciones son sospechosas de que algún proceso no está actuando de manera correcta, aunque no se puede asegurar.

La mayor parte de los problemas puede estar relacionada con un comando en específico y, con frecuencia, se asocia con archivos o directorios que faltan o con permisos incorrectos en archivos existentes.

Ejecución de trabajos en lotes

Un lote de trabajo está formado por un grupo de procesos que se ejecutan uno a continuación de otro. El lote se forma tecleando una secuencia de comandos separados por punto y coma (;), y puede ejecutarse como proceso de superficie o de fondo. Observe la figura IV.8.

```
$(sleep 60; ls -l)&
[1] 1466
$jobs
[1]+  Running  ( sleep 60; ls --color=tty -l ) &
$ps
PID        TTY      TIME          CMD
1349       pts/0 00:00:00         bash
1466       pts/0 00:00:00         bash
1467       pts/0 00:00:00         sleep
1468       pts/0 00:00:00         ps
$kill 1467
$
```

Figura IV.8. Trabajo con lotes desde una terminal.

La secuencia (sleep 60; ls -l)& envía el lote a ejecutarse como una tarea de fondo. El SO devuelve el PID del nuevo proceso (que es el lote entero). Puede matarse el lote entero, como ya se vio, con la sintaxis kill %1, o se puede matar el primer proceso de la lista (sleep en este caso) y observar cómo continúa el segundo proceso. Para hacer eso, habrá que usar el comando ps para que informe el PID del proceso que se desea eliminar. Observe de nuevo la figura IV.8.

IV.4 REDIRECCIÓN DE ENTRADA/SALIDA

En los SO de la familia Unix, existen tres manipuladores de archivos que están siempre abiertos, estos son:

1. La **entrada estándar** o stdin es el teclado y tiene el manipulador 0.
2. La **salida estándar** o stdout es el monitor y tiene el manipulador 1.
3. La **salida de error estándar** o stderr es el monitor y tiene el manipulador 2.

La mayoría de los comandos envía sus salidas hacia la salida estándar (el monitor) y toma las entradas desde la entrada estándar (el teclado). Es posible redirigir esos torrentes de entrada hacia otros lugares, lo cual se puede hacer desde dentro de algún programa en Shell script o desde el propio Shell.

Para redirigir la salida estándar en el Shell, se usan los símbolos > y >>.

* Con el símbolo > la salida estándar se redirige hacia un archivo; si el archivo existe, se sobrescribe y si no existe, se crea.

* Con el símbolo >> la salida estándar también se redirige, pero si el archivo existe, la información nueva se agrega al final del archivo.

Para entender el concepto, se propone el siguiente ejercicio:

1. Redirija la salida del comando ls hacia un archivo en la forma siguiente: ls -l > Dir
2. Ahora redirija la salida del comando cat en la forma siguiente: cat >> Dir

   Después de la orden anterior, el Shell no devolverá el *prompt* debido a que cat está tomando las entradas desde el teclado y se quedará esperando por que se teclee el fin de archivo (<ctrl> <d>). Lo que se escriba se situará al final del archivo Dir.
3. Escriba algo que permita distinguir lo nuevo de lo escrito antes en el archivo Dir (era un listado) y después teclee <ctrl> <d> (fin de archivo en Unix), de modo que se termine de ejecutar el comando cat.
4. Edite el archivo Dir y observe que contiene un listado del directorio más lo que usted le acaba de agregar.

   Para redirigir la entrada estándar en el Shell, se usa el símbolo <. Para seguir con el ejemplo, se puede usar el comando wc (contador de caracteres) para que cuente los caracteres que tiene el archivo Dir que se creó en el ejemplo anterior.
5. Teclee wc < Dir. En este caso, se ha redirigido la entrada por defecto del comando wc (el teclado) para que la tome desde un archivo (Dir). El comando informará la cantidad de caracteres contenidos en el archivo Dir.

IV.5 TUBERÍAS Y FILTROS

Para que dos o más procesos puedan cooperar entre sí, es necesario que se establezca una forma de comunicación.

En el caso de los hilos o procesos ligeros que pertenecen a un mismo proceso pesado, la comunicación se puede lograr a través de la memoria que comparten. Los procesos pesados no comparten memoria y para comunicarse pueden enviarse mensajes.

Conceptualmente, una tubería es una vía de comunicación entre dos procesos. En este tipo de conexión, la salida de un comando se conecta a la entrada de otro. Existen dos tipos de tuberías: con nombre y sin nombre.

El símbolo que se utiliza, en el Shell, para hacer la conexión a través de una tubería sin nombre, es una barra vertical (|) que conecta dos comandos en la forma:

comando1 | comando2.

El código de retorno de la tubería es el código de retorno del último comando. Las tuberías sin nombre se llaman así porque ellas establecen una vía de comunicación temporal que no deja ninguna huella.

Para formar una tubería con nombre, o **FIFO**, se usa el comando mkfifo, el cual crea un archivo (la tubería) que es la vía de comunicación. La utilidad mayor de las tuberías con nombre es cuando se usan dentro de algún programa.

Si se teclea el comando mkfifo tub, se creará una tubería con nombre tub, lo cual se puede apreciar dando la orden de listar el directorio en formato largo (ls -l); el listado obtenido mostrará el archivo tub con una p (*pipe* - tubería) en la primera columna, por ejemplo: prw-r--r--1 mlezcano 0 2015-08-24 17:56 tub.

Para probar la comunicación a través de una tubería con nombre, haga el siguiente ejercicio:

1. Ejecute el comando mkfifo tub, con lo cual creará la tubería con nombre tub (puede usar cualquier otro nombre).
2. Ejecute el comando ls -l y compruebe que se ha creado la tubería con el nombre escogido por usted.
3. Ejecute el comando cat tub. Ese comando se quedará esperando (no devuelve el *prompt*) a que el archivo tub tenga un contenido con fin de archivo.
4. Abra ahora otra terminal y teclee el comando ls > tub, o cualquier otro comando que escriba sobre la tubería.
5. Observe cómo en la primera terminal saldrá el listado que se pidió en la segunda terminal. Con esto se ha establecido una vía de comunicación, a través de la tubería con nombre, entre dos procesos que están en diferentes terminales. Obsérvese que para establecer la comunicación los dos procesos tienen que conocer el nombre de la tubería.

Los filtros están muy asociados al concepto de tubería, aunque no se usan solo en ellas. Un filtro es un programa que toma entradas de otro, le realiza alguna transformación, "la filtra" y la deja seguir.

En Unix, existe una cantidad considerable de utilitarios que se programaron como filtros. Todos los parámetros de un filtro se deben especificar en la línea de comando, dado que ellos no permiten interacción con los usuarios.

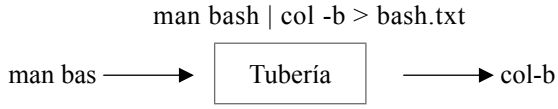


Figura IV.9. Visión esquemática de una conexión a través de una tubería sin nombre.

La figura IV.9 es un ejemplo de combinación de tuberías, filtros y redirección de salida. Vale la pena analizar la orden dada, en la cual se han efectuado varias acciones que se enumeran a continuación:

1. La salida del comando man bash, que saldría por el monitor, se ha enviado a una tubería sin nombre (usando el símbolo de la barra vertical |), o en lenguaje más técnico, la salida de man bash se ha conectado a la entrada o extremo izquierdo de la tubería (figura IV.9).
2. La salida o extremo derecho de la tubería se conectó como entrada del comando col -b.

En resumen, el comando man bash | col –b crea una tubería (sin nombre) que conecta la salida del comando man bash a la entrada del comando col -b, como se aprecia en la figura IV.9.

El comando col es un filtro, es decir, un comando que toma una entrada y la transforma de alguna manera.

3. Por último, la salida del comando man bash | col -b se redirige (usando el símbolo mayor que >) al archivo bash.txt.

Como ya se ha visto, buscar información en el manual puede ser frustrante si no se hace en forma adecuada. Suponga que se desea encontrar toda la información acerca de un tema dado, por ejemplo, acerca de los procesos.

Se puede usar el siguiente comando: man -k process | more, en el cual se hace una búsqueda de la palabra process en todo el manual; como la salida va a ocupar más de una pantalla, se envía (a través de una tubería) al filtro more para que la muestre pantalla a pantalla. Una mejor opción sería usar el filtro less, debido a que ese filtro le permite al usuario retroceder o avanzar con las flechas de dirección.

Otra alternativa sería redirigir la salida hacia un archivo para después poder leerlo con cualquier editor: man -k process > procesos.

La salida tendrá diferentes palabras que algunas veces irán acompañadas de un número entre paréntesis, el cual indica la sección del manual que contiene el tópico, por ejemplo: exit(2) hace referencia a que en la llamada al sistema exit (sección 2 del manual) se trata algo acerca de los procesos.

IV.7 EXPRESIONES REGULARES

Una expresión regular es un **patrón** que define regularidades implícitas sobre cadenas de caracteres alfanuméricos, que permiten representarla en función de otros símbolos (conocidos como **metacaracteres**) de una forma compacta y generalizada.

Las expresiones regulares se usan por muchos programas, por ejemplo, los editores vi y sed, el comando grep, etc. Dichas expresiones permiten realizar diversas operaciones de una forma simple y rápida.

La forma en que se representan las expresiones regulares difiere de un programa a otro, debido a que los conjuntos de caracteres utilizados para definirlas pueden ser distintos, aunque muchas veces coinciden.

Las expresiones regulares son útiles para buscar patrones de cadenas de caracteres y para realizar operaciones de sustitución, entre otras cosas. Para ejemplificar el uso de las expresiones regulares, se utilizará el comando de búsqueda grep, cuya sintaxis es la siguiente:

grep [OPTIONS] PATTERN [FILE…]
grep [OPTIONS] [-e PATTERN | -f FILE [FILE…]

Cuando grep encuentra el patrón buscado, imprime la línea que lo contiene; en algunas versiones de Linux, por ejemplo, Ubuntu, la cadena encontrada se resalta con algún color.

Tabla IV.2.

Metacaracteres usados en expresiones regulares básicas

METACARÁCTER	SIGNIFICADO	EJEMPLOS	INTERPRETACIÓN DEL EJEMPLO (se usa el archivo arc.)
. (un punto)	Representa cualquier carácter en esa posición (solo uno)	grep 'ps.' arc	Busca todas las cadenas **ps** seguidas por cualquier carácter
[c...]	Representa caracteres individuales dentro del símbolo []	grep '[ps0]' arc	Busca todas las cadenas que contengan los caracteres individuales: **p**, **s** o **0**
[c1-c2]	Representa un carácter dentro del rango c1-c2	grep '201[0-6]' arc	Busca todas las cadenas que contengan números en el rango **2010-2016**
[^cadena]	Niega los caracteres de la cadena de caracteres	grep '[^ps0]' arc	Busca todas las cadenas que contengan caracteres diferentes a: **p**, **s** o **0**
<	Ningún carácter a la izquierda de la cadena especificada	grep '\<passwd' arc	Busca todas las cadenas **passwd** sin ningún carácter a la izquierda, de forma que **apasswd** no machea
>	Ningún carácter a la derecha de la cadena especificada	grep '\<passwd\>' arc	Busca todas las palabras **passwd**
^, $	Anclaje al principio o al final de la línea, respectivamente	grep '^La' arc grep 'que$' arc	Busca todas las líneas que comienzan por **La** y todas las que finalizan con **que**

Tabla IV.3.

Metacaracteres usados en expresiones regulares extendidas

METACARÁCTER	SIGNIFICADO	EJEMPLO	INTERPRETACIÓN DEL EJEMPLO
?	El carácter que le antecede es opcional	grep -E 'psa?' arc	Busca todas las cadenas que contengan a **ps** seguidas o no por **a**
|	Una cadena o la otra (*or*)	grep -E 'ps|2015' arc	Busca todas las cadenas que contengan a **ps** o **2015**
+	El carácter de la izquierda existe una o más veces	grep -E 'ab+' arc	Busca todas las cadenas que contengan la letra **a** seguida por una o más letras **b**
*	El carácter de la izquierda existe 0 o más veces	grep -E 'ab*' arc	Busca todas las cadenas que contengan la letra **a** seguida por la letra **b** 0 o más veces
{n1} {n1, n2}	Define la cantidad de repeticiones o un rango; n1 y n2 son números enteros	grep -E '\<[0-9]{3}>' arc grep -E '\<[0-9]{2, 3}>' arc	1. Busca todas las cadenas que contengan tres dígitos decimales 2. Busca todas las cadenas que contengan dos o tres dígitos decimales

Las tablas IV.2 y IV.3 muestran algunos metacaracteres que son bastante generales (se usan en muchas aplicaciones); en los ejemplos, se usa el comando grep y todas las expresiones regulares se han puesto entre apóstrofos. Esto no siempre es necesario, pero a veces sí, por lo cual es mejor acostumbrarse a usarlas de esa forma; las búsquedas se hacen dentro del archivo arc.

La tabla IV.2 usa las expresiones regulares básicas, mientras que la tabla IV.3 usa las expresiones regulares extendidas, por eso en esta última se usa el comando grep con la opción E (*extended*).

Un carácter cualquiera o una palabra son expresiones regulares por sí mismas; por ejemplo, si se desea buscar la palabra ps dentro de un archivo que se llama fileTest, se puede usar el comando grep en la forma siguiente: grep ps fileTest. En este caso, ps es la expresión regular que solo coincide (machea) con un patrón que sea exacto a esa palabra.

En algunos casos, es necesario usar el carácter de escape \[55] para que un determinado metacarácter no se interprete literalmente como el símbolo que es; por ejemplo, el comando grep '\<passwd\>' arc busca la palabra passwd (espacios a su izquierda y derecha), mientras que grep '<passwd>' arc busca la cadena <passwd>, debido a que los símbolos < y > se interpretan literalmente en este último caso.

IV.8 PATRONES DE ARCHIVOS

Los patrones de archivos, comodines o *wildcards*, se usan para ejecutar acciones sobre el sistema de archivo.

Muchas personas confunden los patrones de archivos con las expresiones regulares, sobre todo por el uso de símbolos semejantes.

El Shell interpreta los patrones de archivos o comodines expandiendo su interpretación, y la acción que se lleve a cabo dependerá del comando que la use; por ejemplo, si el comando es ls, se imprime un listado de todos los archivos que respondan al patrón y si es rm, se borran esos archivos.

La tabla IV.4 muestra un resumen de los caracteres que se usan como comodines. Obsérvese que en este contexto los símbolos (*, +, etc.) tienen un significado distinto al que tienen en las expresiones regulares.

También debe observarse que las acciones se hacen sobre el sistema de archivo y no sobre textos (como los hacen las expresiones regulares).

55 Cuando el Shell se encuentra el carácter \, interpretará el carácter que le sigue de forma especial.

Tabla IV.4.

Comodines en los Shell de Unix (los ejemplos suponen que el directorio actual contiene los archivos: tap, tarea, tub, tuber, tv, ar)

COMODÍN	SIGNIFICADO	EJEMPLO	INTERPRETACIÓN DEL EJEMPLO
?	Cualquier carácter en esa posición.	ls -l t?b El archivo tub está representado en ese patrón.	Listar todos los archivos cuyos nombres tienen exactamente tres caracteres: primero una **t**, después cualquier carácter y tercero una **b**.
*	Todos los caracteres a partir de esa posición.	cp t?b* /home Los archivos tub y tuber están representados en ese patrón.	Copiar todos los archivos que responden al patrón, desde el directorio actual hacia el directorio /home
[c1-c2]	Cualquier carácter simple en el rango definido entre el carácter c1 y el carácter c2.	mv t[a-u]* /usr/C Los archivos tap, tarea, tub, y tuber están representados en ese patrón.	Mover todos los archivos que responden al patrón, desde el directorio actual al directorio /usr/C
!	Niega la expresión que le sigue.	rm t[!a-u]* El archivo tv está representado en ese patrón.	Borrar los archivos que responden al patrón: aquellos cuyos nombres comienzan por **t** y cualquier carácter en la segunda posición que no esté en el rango **a - u**
{c1,...,cn}	Define un conjunto de caracteres c1, c2, ...cn.	ls -l {a,t}* Los archivos tap, tarea, tub, tuber, tv, ar machean con ese patrón. ls -l t{a,t}* La salida estará formada por los archivos tap y tarea. Además, se verá lo siguiente: ls cannot access tt*	En el primer ejemplo, se listan todos los archivos cuyos nombres comienzan por el carácter **a** o por el carácter **t**. En el segundo ejemplo, se expande una interpretación **tt** que no existe y por eso se emite el mensaje de error.

IV.9 RESUMEN DEL CAPÍTULO

Los SO de la familia Unix, o dicho de otra forma, al estilo Unix, se han difundido por todo el mundo y existen diversas distribuciones que tienen características particulares. A pesar de esa diversidad, todas las distribuciones mantienen rasgos comunes que permiten estudiarlas en una forma genérica.

Los sistemas tipo Unix son sistemas:

* Multiusuarios, porque permiten que muchos usuarios estén conectados a un mismo sistema central desde diversas terminales (de distintos tipos).
* Multiprocesamiento, porque son capaces de explotar más de un procesador.
* Multitarea, debido a que pueden correr más de una tarea en cada uno de los procesadores.

Cuando se instala un SO de esta familia, se puede escoger el intérprete de comandos o Shell a instalar, incluso puede ser uno propio. Algunos de los intérpretes de comandos más comunes son el Bourne (sh), el Bourne Again Shell (bash) y el C Shell (csh).

El Shell no solo es una interfaz de ejecución de comandos, sino que también es una poderosa herramienta que maneja distintos tipos de variables y formas de interactuar con los procesos.

Los procesos se comunican entre sí a través de señales y existe una amplia diversidad de ellas.

IV.10 EJERCICIOS PROPUESTOS

Para resolver algunos de los ejercicios que se proponen a continuación, será necesario que busque ayuda acerca de los comandos involucrados. Use el comando man.

1) Entrada al sistema. Desde un SO de la compañía Microsoft, ejecute el programa putty.exe.
   a) Conéctese de forma remota a una estación Unix usando el protocolo **SSH** (puerto 22).
   b) Introduzca su nombre de usuario y su contraseña. Explique el significado de la información inicial que muestra el sistema.
2) Cambiar la contraseña. Use el comando passwd para cambiar su contraseña. ¿Qué requisitos debe tener la contraseña para que sea segura?
3) Uso del comando man. Para obtener ayuda sobre un comando, use el comando man.
   a) Observe cómo está organizada la ayuda y cómo trabajar con ella, teclee: man man.
   b) Busque el campo See also. ¿Qué significa el número entre paréntesis?
4) Otra forma de obtener información es con el comando info. Teclee man info para ver cómo se usa.
5) Use el comando date para ver la fecha y la hora.
6) Use el comando tty para ver desde qué terminal está usted trabajando.
7) Use el comando who para ver todos los usuarios que están conectados.
8) Use el comando who am i para saber su nombre de usuario. ¿Qué otra información se muestra?
9) Use el comando who -u, ¿cuál es la diferencia con el anterior?
10) En este ejercicio se explora el sistema de archivos.

a) Vaya a su directorio origen (home directory). ¿Cuál es la forma más inmediata de acceder a ese directorio a través de comandos?

Sugerencia: Use el comando cd.

b) Determine el directorio de trabajo actual. Observe que el directorio origen y el de trabajo no tienen que ser los mismos, debido a que mientras el directorio origen no cambia en toda la sesión, el directorio de trabajo cambiará cada vez que usted se cambie de directorio.

c) Cámbiese al directorio /etc y liste los archivos del nuevo directorio de trabajo.

d) Cámbiese al directorio raíz y liste los archivos. Trate siempre de cambiarse de un directorio a otro usando la menor cantidad de órdenes posibles.

e) Vuelva al directorio de inicio y liste (comando ls) los archivos de los directorios /bin y /usr sin cambiar de directorio de trabajo.

f) Liste todos los archivos del directorio actual, incluyendo los archivos ocultos (use el formato largo). Explique el significado de cada columna y los permisos de acceso que existen.

11) En este ejercicio se trabaja con los contenidos de los archivos.

   a) Usando el comando cat, muestre el contenido del archivo /etc/passwd.

   b) El contenido del archivo /etc/passwd es demasiado grande y por eso no se pudo ver todo, ahora muéstrelo pantalla a pantalla de forma que pueda verlo más detenidamente.

      Sugerencia: use una tubería y un filtro.

      i) Explique qué es una tubería.

      ii) Explique qué es un filtro.

   c) Cree un archivo en su directorio origen.

      Sugerencia: use el comando cat y la redirección de salida.

      i) Explique qué es la redirección de salida, ¿existe algún otro tipo de redirección? Explique.

12) En este ejercicio se trabaja con información del sistema de archivos.

   a) Analice el uso que el sistema de archivos le da al espacio en disco.

      Sugerencia: Use el comando df en la forma df -Ph.

   b) ¿Cómo se nombran los sistemas de archivos? ¿Qué significa el lugar donde se monta cada uno?

   c) Liste la información sobre la utilización de los nodos-i (*i-node*), use df -i.

      i) Explique qué son los nodos-i.

d) Vaya al directorio origen y determine el espacio en disco ocupado por cada directorio, use du -h.

13) En este ejercicio se manipulan archivos.

a) Vaya al directorio origen, copie un archivo cualquiera hacia el directorio origen.

b) Cree (comando mkdir) un nuevo directorio llamado temp.

c) Liste el contenido del directorio de trabajo usando la forma ls -F, analice la salida.

d) Mueva (comando mv) un archivo cualquiera al directorio de trabajo. Explique la diferencia entre copiar y mover.

e) Cambie el directorio de trabajo hacia temp.

f) Copie (comando cp) un archivo cualquiera hacia el directorio de trabajo de modo que quede con el nuevo nombre myfile.a.

g) Cambie el nombre del archivo myfile.a por myfile.b.

h) Liste los nombres de archivos usando comodines.

i) Borre los archivos del directorio temp que comienzan con m.

j) Borre el directorio temp y verifique que se ha borrado.

Los ejercicios 14 al 25 trabajan con procesos.

14) Ejecute el comando ps.

a) Explique el objetivo general del comando y de cada una de sus opciones.

b) Interprete el significado de las columnas de la salida.

c) Pruebe los ejemplos dados en la página del manual del comando ps. Compárelos.

d) Haga una lista de todos los procesos que se ejecutan a su nombre, mostrando el proceso padre (sugerencia: use la opción -l).

e) ¿Quién es el proceso padre de ps?

15) Liste todos los procesos en forma de árbol (sugerencia: use pstree y explore las opciones).

a) ¿Quién es el padre de todos los procesos?

Sugerencia: para ver resaltada la rama que le corresponde, use la opción -h.

16) Ejecute el comando ls -l -R / >dir 2>error como proceso de fondo (*background*).

a) ¿Qué hace este comando?

17) Ejecute el comando compuesto (sleep 240; ls -l > listing) como proceso de fondo.

a) ¿Qué hace este comando?

18) Liste los procesos que se están ejecutando (comando ps) y los trabajos (comando jobs). Compare la información que ofrece cada uno y diga las diferencias entre trabajo (*job*) y proceso.

19) Ejecute el comando compuesto (sleep 120; who) como proceso de fondo.

a) Mate el proceso correspondiente a sleep del ejercicio 17. Use el comando kill. ¿Qué sucedió? Explique.

Advertencia: Debe tener cuidado al usar el comando kill, pues cuando se elimina un proceso, puede perderse el trabajo hecho.

20) Repita el ejercicio 17. Ahora, mate el trabajo resultante usando la sintaxis:

```
kill % <id del trabajo>
```

Nota: Si un proceso no se deja matar con kill [id del proceso], use kill enviando la señal TERM en la forma siguiente: kill -9 [id del proceso].

21) Ejecute el siguiente comando:

```
find / -type f -name test >busqueda 2>/dev/NULL
```

a) Suspenda el proceso anterior presionando ctrl z.

b) ¿Qué información le ofreció el sistema? ¿Qué significa que el proceso está suspendido?

Si el proceso anterior ejecuta muy rápido, repita el ejercicio con el comando: ls -R / >busqueda 2>/dev/NULL

22) Continúe el proceso anterior como proceso de fondo usando el comando bg.

a) Mueva el proceso anterior a la superficie usando el comando fg.

b) Para recuperar el *prompt*, vuelva a enviar el proceso anterior al fondo.

c) Mate el proceso anterior.

23) Liste los procesos que están corriendo en otra terminal.

24) Ejecute el comando sleep 120 y asuma que se bloqueó y perdió todo control de la terminal. ¿Cómo lo puede matar?

En los ejercicios del 26 al 30 se trabaja con variables de entorno. Las variables de entorno tienen asociadas cadenas que determinan el comportamiento de muchos programas y funciones.

25) Ejecute el comando set. ¿Qué información se muestra?

26) Usando el comando echo, muestre en pantalla el contenido de la variable PATH.

a) Explique el uso de esta variable de ambiente.

b) Añada una nueva entrada a la variable PATH y después observe el resultado; escriba: PATH="$PATH:/usr"; echo $PATH

27) Muestre en pantalla la variable HOSTNAME. Diga cuál es su significado y para qué se usa.

28) El *prompt* primario es el símbolo que muestra el Shell para informar que está esperando órdenes. En el **Bash**, ese símbolo se almacena en una variable de entorno llamada PS1. Para cambiar el *prompt*, use el comando:

PS1="<string>". La cadena <string> puede contener algunos símbolos que el Shell interpreta de forma especial:

\d muestra la fecha.

\h muestra el nombre del host.

\n inicia una nueva línea.

\u muestra el nombre de usuario actual.

\w muestra el directorio de trabajo.

a) Muestre en pantalla el contenido de la variable PS1.

b) Cambie el *prompt* para que tenga la forma a -->

c) Cambie el *prompt* para que muestre su directorio de trabajo seguido del símbolo de dólar ($).

29) Termine la sesión y comience una nueva.

a) ¿Se salvaron los cambios hechos en el *prompt*? Explique.

30) Cree los siguientes alias y compruébelos:

```
dir=ls -l
rename=mv
rm=rm -i
```

31) La opción noclobber del comando set previene que se sobrescriba sobre archivos. Teclee: set -o y analice las opciones.

a) Use la redirección de salida para crear un archivo nuevo.

b) Active la opción noclobber para que no se puedan sobreescribir los archivos que existen, use la forma set -o noclobber.

c) Vuelva a redirigir la salida hacia el archivo creado en el inciso a. ¿Explique qué sucede?

d) Desactive la opción noclobber, use la forma set +o noclobber. Ahora intente hacer el inciso c, ¿qué sucede? Explique.

32) Edite los archivos .bash_profile y .bashrc. ¿Para qué se usan estos archivos?

33) Configure sus archivos .bash_profile y .bashrc para personalizar su cuenta con las opciones antes vistas. Observe que ambos archivos son: **archivos puntos**, que tienen el atributo de ser ocultos, y archivos de recursos o **rc**.

34) ¿Qué salida ocasionaría cada una de las siguientes órdenes si se ordenara consecutivamente?

a) set a b c d e f g h i j k l m n

b) echo $10

c) echo $15

d) echo $*

e) echo $#

f) echo $?

g) echo ${11}

35) Explique el resultado de cada una de las siguientes órdenes, ejecutadas en secuencia:

a) B=ls

b) A=$B

c) echo $A

d) eval $A

36) Asigne el último parámetro posicional a la variable ULT.

37) Imprima por pantalla las variables de entorno del Shell, de modo que aparezcan página a página.

38) Almacene en un archivo llamado entorno.txt las variables de entorno de su sesión. ¿Cuántas variables hay definidas?

39) Utilice la instrucción uname para obtener la versión del SO. Utilice el manual para usar algunas opciones de uname.

40) El comando lsb_release imprime información útil acerca de LSB (Linux Standard Base); si tiene un sistema Linux, úselo y explore sus posibilidades.

a) Use las opciones -a, -i.

IV.11 BIBLIOGRAFÍA CONSULTADA

Garrels, M. *Bash guide for beginners*. Recuperado el 24 de febrero de 2016, de http://tille.garrels.be/training/bash/

Gilly, D. (2003). *Unix in a Nutshell*. En *The Unix CD Bookshelf* (3.ª ed.). Sebastopol: O'Reilly Media.

Love, P., Merlino, J., Red, J. C., Zimmerman, C., & Weinstein, P. (2005). *Beginning UNIX*. Nueva Jersey: John Wiley & Sons.

Mohr, J. (2001). *Linux-user's resource*. Nueva Jersey: Prentice Hall.

Peek, J., O'Reilly, T., & Loukides, M. (2003). *Unix Power Tools*. En *The Unix CD Bookshelf* (3.ª ed.). Sebastopol: O'Reilly Media.

Peek, J., Todino, G., & Strang, J. (2003). *Learning the Unix Operating System*. En *The Unix CD Bookshelf* (3.ª ed.). Sebastopol: O'Reilly Media.

Petersen, R. (2009). *Manual de referencia Linux* (6.ª ed.). Nueva York: McGraw-Hill.

Robbins, A., & Lamb, L. (2003). *Learning the vi editor*. En *The Unix CD Bookshelf* (3.ª ed.). Sebastopol: O'Reilly Media.

Robbins, K., & Robbins, S. (2003). Unix Systems Programming: Communication, concurrency, and threads. Nueva Jersey: Prentice Hall.

Rosenblatt, B. (2003). *Learning the Korn Shell*. En *The Unix CD Bookshelf* (3.ª ed.). Sebastopol: O'Reilly Media.

Shotts Jr., W. (2012). The Linux Command Line: A complete introduction. San Francisco: No Starch Press.

Tiemann, B., & Urban, M. C. (2001). *FreeBSD Unleashed*. Indianápolis: Sams Publishing.

Cooper, M. *Advanced Bash-Scripting Guide. An in-depth exploration of the art of shell scripting*. Recuperado el 24 de febrero de 2016 de http://www.tldp.org/LDP/abs/abs-guide.pdf

CAPÍTULO V

Programación en Shell script

RESUMEN

Este capítulo aborda el lenguaje Shell script como herramienta importante para automatizar muchas tareas que deben realizar los administradores de los sistemas operativos. El hecho de que este poderoso lenguaje acompañe a todos los sistemas operativos tipo UNIX enfatiza la importancia de los contenidos que se discuten en esta parte del libro. El capítulo presenta los contenidos de la forma más práctica posible, y por ese motivo, casi todos los aspectos discutidos se acompañan por ejercicios que enfatizan en la programación y su uso desde la perspectiva de los sistemas operativos. Se analizan las estructuras más importantes del lenguaje Shell script, lo que incluye: los operadores permitidos, las sentencias condicionales y de repetición, y también se dedica un espacio para discutir el uso y la forma de trabajo con arreglos y funciones. El capítulo finaliza con un resumen, y es el único que no tiene una sección de ejercicios propuestos al final, debido a que en todo su contenido se presentan y analizan diversos programas. En lugar de presentar una sección de ejercicios para este capítulo, el autor ha preferido emitir algunas recomendaciones que son en sí propuestas de ejercicios que el estudiante deberá aceptar para demostrarse a sí mismo que ha asimilado los contenidos discutidos, no solo en este capítulo sino también en el resto del libro.

Palabras clave: Shell script, script, lenguajes de programación, sentencias, arreglos, funciones.

¿Cómo citar este capítulo? / How to cite this chapter?

Lezcano-Brito, M. G. (2017). Programación de Shell script. En *Fundamentos de sistemas operativos. Entornos de trabajo* (pp. 171-211). Bogotá: Ediciones Universidad Cooperativa de Colombia.

CHAPTER V

Shell scripting

ABSTRACT

This chapter addresses the shell script language as an important tool for automating many tasks that must be performed by operating system administrators. The fact that this powerful language accompanies all UNIX-like operating systems highlights the importance of the contents reviewed in this part of the book. The chapter presents the contents in the most practical way possible, and therefore, almost all aspects covered are accompanied by exercises on programming and its use from the perspective of operating systems. The most important structures of the shell script language are explored, such as allowed operators, conditional and repetitive statements, while explaining how to use and work with arrays and functions. The chapter ends with a summary and is the only one that does not contain a section of proposed exercises since multiple programs are presented and analyzed throughout its contents. Instead of presenting a section of exercises for this chapter, the author has preferred to give some recommendations that are, in essence, proposed exercises that the student must accept to prove to himself that he has assimilated the contents discussed, not only in this chapter but also in the rest of the book.

Keywords: arrays, functions, programming languages, shell script, script, statements.

V.1. LOS LENGUAJES DE PROGRAMACIÓN Y EL SHELL SCRIPT

Los idiomas o lenguajes humanos sirven para que las personas se comuniquen entre sí; así mismo, cuando la comunicación es escrita, existe un conjunto de reglas sintácticas que especifican la forma correcta en que deben escribirse las palabras y un conjunto de reglas semánticas que sirven para interpretar lo que se escribe.

Los lenguajes de programación son una forma especial de comunicación entre el hombre y la computadora, y asociados a ellos también existen reglas sintácticas y semánticas que los definen formalmente.

El objetivo de los lenguajes de programación es darle órdenes adecuadas al equipo de cómputo para que realice diversas tareas. Existen diferentes clases o tipos de lenguajes de programación. El más básico es el **lenguaje de máquina**, que utiliza el alfabeto binario para escribir las instrucciones en el propio código de la máquina y por eso no necesita ninguna traducción.

Hay un grupo de lenguajes que se conoce como de **bajo nivel**, debido a que la forma en que se codifican las instrucciones está muy cercana al lenguaje de la máquina, un ejemplo es el ensamblador. Este tipo de lenguaje también es específico de un solo tipo de máquina y necesita un traductor que transforme el código escrito al código de la máquina sobre la cual se ejecutará el programa.

Otro grupo de lenguajes se considera de **alto nivel**, debido a que son independientes de la máquina y se acercan más al lenguaje humano. Este tipo de lenguaje necesita un intérprete[56] o compilador que traduzca las instrucciones para que puedan ejecutarse en la máquina. Existen muchos lenguajes de programación que se pueden agrupar en esta categoría.

A su vez, pueden encontrarse muchas otras formas de agrupar los lenguajes de programación, pero no es el objetivo de este libro discutir el tema y solo se han mencionado estas clasificaciones generales para introducir el lenguaje Shell script.

Como ya se vio en el capítulo anterior, los Shell de los sistemas Unix permiten interpretar y ejecutar diversas órdenes. La orden más "compleja" que se analizó en ese capítulo fue enviar un conjunto de comandos al Shell para que lo ejecutara como un proceso de fondo, la orden mencionada tiene la forma (comando1; comando2; comando3; comando4)&.

56 Intérprete. Programa informático que analiza y ejecuta, una a una, las instrucciones de un programa escritos en un lenguaje de alto nivel. No produce código de máquina como los compiladores.

Esas mismas órdenes (no necesariamente para que se ejecuten de fondo) se pueden introducir dentro de un archivo de texto y pasarle el nombre del archivo al Shell para que las **interprete** y las ejecute (el Shell es un intérprete).

Entonces, en su forma más sencilla, un programa en Shell script no es más que un conjunto de comandos ordinarios contenidos dentro de un archivo de texto plano. Los Shell de los SO de la familia Unix son más complejos que eso y pueden, entre otras cosas: interpretar comandos compuestos, evaluar expresiones que contienen diferentes operadores, ejecutar subcomandos que se asemejan a sentencias de cualquier lenguaje de programación, trabajar con funciones definidas, contener comentarios, trabajar con parámetros, manejar arreglos, sustituir comandos, etc. En fin, toda una gama de facilidades que están presentes en los lenguajes de alto nivel.

Justificación del uso del lenguaje

Con el lenguaje Shell script, se pueden crear comandos propios que se adecuen a las necesidades de cada usuario, lo que permite automatizar muchas tareas rutinarias y lograr una mejor inversión del tiempo. Lo bueno es que la programación del nuevo comando se puede auxiliar directamente de los comandos que se agregan cuando se instala el SO y de los que han hecho otros colegas, todo lo cual se logra relativamente fácil y rápido.

Los administradores de sistema pueden automatizar sus labores e intercambiar con otros colegas los comandos que les han resultado útiles, de forma que se provean de una amplia gama de herramientas que les ayuden a hacer muchas tareas rutinarias que poco a poco irán automatizando.

Para este capítulo, se ha escogido el lenguaje script que acompaña al Bash (Bourne Again Shell), debido a que es compatible con el del Bourne Shell e incorpora facilidades del Korn Shell y el C Shell. Además, en muchas distribuciones de la familia de SO Unix se usa el Bash o el Bourne como Shell por defecto. Los contenidos abordados se presentarán en la forma más práctica posible.

V.2. GENERALIDADES

En general, para que un archivo se pueda ejecutar deberá tener permiso de ejecución. Observe la figura V.1, que muestra el listado de un directorio obtenido con el comando ls.

```
$ls -l
total 20
_  rwx------  madiedo  Master   1024  jul  8  12:49  cpu.c
d  rwxr-xr-x  lezcano  root     4096  jun  1  20:44  trabajo
l  rwx------  madiedo  Master    512  sep  2  12:56  archivos
_  rwx------  lezcano  root     1024  sep  1  23:36  file1
_  rwx------  lezcano  root     4096  sep  2  14:53  clase1
_  rwxr-xr--  lezcano  root     4096  sep  2  14:53  Example
$
```

Figura V.1. Listado en formato largo.

En los SO Unix, los permisos sobre los archivos se establecen de acuerdo con tres tipos de usuarios: el propietario (*owner*), los miembros del grupo (*group*) y los demás usuarios (*others*).

La segunda columna de la figura V.1 muestra una cadena o tira de nueve caracteres (internamente son nueve bits), que define los permisos asociados a cada archivo: los primeros tres campos corresponden a los permisos del **propietario** (*owner*), los tres campos que siguen definen los permisos que tienen los miembros del grupo (*group*) y los tres últimos campos son los permisos de los demás usuarios (*others*), es decir, de aquellos que no son dueños del archivo, ni miembros del grupo asociado al proceso (en Unix es obligatorio que los usuarios pertenezcan al menos a un grupo).

Cuando se lista un directorio, los tres campos de cada terna, dentro de la tira de nueve bits, pueden mostrar:

* El primer elemento de la terna: una letra **r** o un guion (-).
  - Si es **r**, se tiene permiso de lectura; si es guion (-), no se tiene permiso de lectura.
* El segundo elemento de la terna: una letra **w** o un guion (-).
  - Si es **w**, se tiene permiso de escritura; si es guion (-), no se tiene permiso de escritura.
* El tercer elemento de la terna: una letra **x** o un guion (-).
  - Si es **x**, se tiene permiso de ejecución; si es guion (-), no se tiene permiso de ejecución. El permiso **x**, es decir, de ejecución, sobre un directorio significa que se puede buscar en él.

Para entender mejor el concepto, se puede analizar el archivo **Example** de la figura V.I. En ese caso, la tira de nueve bits tiene la forma: **rwxr-xr--**, es decir que las tres ternas son:

1. La primera, **rwx**, especifica que el propietario puede leer (**r**) el archivo, escribir (**w**) sobre él y ejecutarlo (**x**).
2. La segunda, **r-x**, les da permiso de lectura (**r**) y ejecución (**x**) a los miembros del grupo, pero no les da permiso de escritura (**-**).
3. La tercera, (**r--**), solo admite el permiso de lectura para los demás usuarios.

El comando chmod permite cambiar esos permisos. La sintaxis de este y otros comandos se debe ver usando la ayuda (man chmod), debido a que pueden diferir de una versión a otra del SO. En este capítulo, se usa la forma sintáctica de la versión GNU[57].

57 GNU, acrónimo recursivo de *GNU is not Unix*, una forma que tiene el movimiento de *software* libre de desligarse de los productos Unix que son propietarios. Los SO GNU/Linux tienen un núcleo, llamado Linux, más una colección de programas GNU (aunque es habitual que se le llame Linux a todo).

Uso de la forma simbólica para cambiar permisos

La figura V.2 muestra el uso del comando chmod para cambiar los permisos de escritura sobre el archivo Example, la orden chmod +w Example les asigna (+) permiso de escritura a todos los usuarios, es decir, al propietario (ya lo tenía), al grupo y a los otros, lo cual se puede apreciar en el listado ordenado después de la ejecución del comando. Obsérvese que la tira de permiso para ese archivo es ahora: **rwxrwxrw-**. La forma se denomina simbólica porque usa letras para especificar los permisos (r, w, x).

```
$chmod +w Example
$ls -l
total 20
_  rwx------   madiedo   Master    1024   jul  8  12:49  cpu.c
d  rwxr-xr-x   lezcano   root      4096   jun  1  20:44  trabajo
l  rwx------   madiedo   Master     512   sep  2  12:56  archivos
_  rwx------   lezcano   root      1024   sep  1  23:36  file1
_  rwx------   lezcano   root      4096   sep  2  14:53  clase1
_  rwxrwxrw-   lezcano   root      4096   sep  2  14:53  Example
$
```

Figura V.2. Cambiando permisos con chmod.

Otra forma, menos abarcadora, de cambiar los permisos es especificando a quién se le desea cambiar ese permiso, para lo cual se usan las letras: **u**, **g**, **o** y **a**, a fin de hacer referencia al propietario (u), al grupo (g), a los demás usuarios (o) y a todos (a). Además del símbolo + para agregar permisos, que se utilizó antes, se usa el símbolo - para quitarlos, por ejemplo:

* chmod u+x file le da al propietario permiso de ejecución sobre el archivo file.
* chmod u-x file le quita al propietario el permiso de ejecución sobre el archivo file.
* chmod g+x file agrega permiso de ejecución sobre el archivo file para el grupo.
* chmod o-x file les quita el permiso de ejecución sobre el archivo file a todos los usuarios que no son el propietario ni un miembro de su grupo.

Existen otros permisos que se pueden fijar utilizando el comando chmod, que son: SUID, SGID y el *sticky bit*.

* Permiso SUID (**S**et **U**ser **ID**). La facilidad SUID define un permiso especial para que un usuario que no es el dueño de un archivo lo pueda ejecutar con los mismos permisos de su propietario. A fin de establecer este permiso, se usan las formas u+s y u-s para agregar y quitar el permiso, respectivamente; obsérvese la referencia al propietario (u).
* SGID (**S**et **G**roup **ID**). SGID es similar a SUID, pero en este caso se refiere a los permisos de ejecución del grupo, que se pueden cambiar para que el usuario ejecute el archivo con los permisos del grupo (como si fuera miembro de él). Si el permiso SGID se le aplica a un directorio, los archivos creados en ese directorio pertenecen al grupo

del cual el directorio es miembro y no al grupo del usuario, esta facilidad puede ser útil para compartir directorios. A fin de establecer el permiso, se usan las formas g+s y g-s para agregar y quitar el permiso, respectivamente; obsérvese la referencia al grupo (g).

* *Sticky bit.* Principalmente, se usa sobre directorios compartidos y permite que cualquiera pueda escribir o modificar un archivo o directorio, pero solo su propietario o el *root* pueden eliminarlo.

A continuación, se muestran algunos ejemplos con estos permisos.

1. Si en una terminal se tecleara algo como ls -l /usr/bin/passwd, el comando ls mostraría lo siguiente:

   -rwsr-xr-x root root 53112 nov 19 16:35 /usr/bin/passwd

   Se puede apreciar que el archivo /usr/bin/passwd tiene establecido el permiso SUID, lo cual se señala con una letra s en el lugar del bit de ejecución del propietario (rws).

2. Ahora supóngase que en el directorio de trabajo existe un archivo, nombrado file, con los permisos siguientes: rw-r--r-- file, y se ejecuta la secuencia de comandos: chmod u+s file; ls -l file.

   El listado obtenido será: -rwSr--r-- file. Nótese que ahora el bit de ejecución del propietario tiene la letra S, porque se ha fijado el permiso SUID; sin embargo, se muestra la letra en mayúscula y no en minúscula como sucedió con el archivo /usr/bin/passwd.

3. Ahora ejecute la secuencia siguiente: chmod u+x file; ls -l. El listado obtenido será: -rwsr--r--file, donde la letra S se ha sustituido por s, lo que significa que ahora el usuario tiene el permiso de ejecución sobre el archivo file, a pesar de no ser el propietario.

La conclusión es la siguiente:

* Si se fija el permiso SUID sobre un archivo que no tiene permiso de ejecución para el propietario (como es el caso del archivo file en el paso 2), no se le está cambiando el permiso de ejecución, sino que se le está agregando el permiso SUID, y por eso el listado lo distingue poniendo la letra S en el bit de ejecución del propietario (rwS), lo cual significa que a pesar de tenerse ese permiso no se podrá ejecutar el archivo, dado que ni siquiera el propietario lo puede hacer.
* De otra parte, si se fija el SUID sobre un archivo que tiene permiso de ejecución para el propietario, el SO le otorga ese permiso al usuario que lo cambió (sin ser el propietario) y lo distingue en el listado con la letra s; es lo que sucede en el paso 3.

El procedimiento usado en los pasos 2 y 3 del ejemplo anterior también es válido con respecto al permiso SGID, pero en ese caso las órdenes se refieren al grupo y no al

propietario, por eso se usa la forma chmod g+s o g-s. Debido a esto, si se da la orden chmod g+s file; ls -l file, después de haber hecho el paso 3 del ejemplo anterior el listado obtenido mostrará los permisos en la forma rwsr-Sr-- file, lo que significa que se ha establecido el permiso SGID sobre el archivo file, pero no tendrá efecto porque ni siquiera los miembros del grupo pueden ejecutar ese archivo. Si después se da la orden chmod g+x file; ls -l file, el listado obtenido será: -rwsr-sr--file, donde se aprecia que ahora el usuario puede ejecutar el archivo como si fuera un miembro del grupo al cual pertenece el archivo file.

Para fijar o quitar el *sticky bit*, se usan los signos + y -, respectivamente, seguido por la letra t, observe el siguiente ejemplo:

1. Se comienza con un archivo, nombrado file, sobre el cual no se tiene ningún permiso, es decir que un listado ls –l file mostraría: --------- file.
2. Ahora se le fija el *sticky bit*: chmod +t file; ls -l y el listado queda en la forma:

   ---------T file, donde se puede observar que la tira de bits de permiso finaliza con una T, lo que significa que se ha fijado el *sticky bit*, pero no tendrá efecto debido a que ningún usuario tiene permiso de ejecución.
3. Ahora se ejecuta la siguiente secuencia: chmod +x file; ls –l, el listado obtenido muestra la forma: --x--x--xt file, donde todos los usuarios tienen el permiso de ejecución y por eso la letra T ha pasado a ser t, porque se ha activado el *sticky bit.*

Forma numérica para cambiar permisos

Las formas para cambiar permisos vistas hasta el momento se conocen como formas simbólicas, debido a que usan letras. Otra manera de hacer lo mismo es usando un número octal que representa un patrón de bits.

El número tiene cuatro dígitos y si falta alguno, se asume que es cero, lo que implica que no se cambia ese permiso, dado que sería un patrón 000. Recuérdese que las tiras rwx son realmente patrones de tres de bits, por ejemplo: rwx es 111, mientras que r-x es 101.

El primer dígito octal corresponde a los permisos SGID, SUID y al *sticky bit*; el segundo representa los permisos del propietario del archivo; el tercer dígito es para los permisos del grupo, y el cuarto es para los demás usuarios. Por ejemplo:

* chmod 7 file es equivalente a chmod 0007 file. En este caso, sobre el archivo file:
  - No se establecen los permisos SUID, SGID ni el *sticky bit* que se corresponden con el primer dígito (el cero en la posición 1 es 000).
  - No se la da ningún permiso al propietario, ni al grupo (los 0 en las posiciones 2 y 3 son 000, respectivamente) y se les dan todos los permisos a los otros usuarios (7 es 111).
* chmod 754 file es equivalente a chmod 0754. En este caso, sobre el archivo file no se establecen los permisos SUID, SGID ni el *sticky bit* que se corresponden con el

primer dígito, se le dan todos los permisos al propietario (7 es 111), al grupo se le asignan los permisos de lectura y ejecución (5 es 101), y a los demás usuarios se les otorga solo el permiso de lectura (4 es 100).

* chmod 7111 file; el primer dígito (7: 111) fija los bits SUID, SGID y el *sticky bit*; los tres dígitos restantes (todos 1: 001) fijan el permiso de ejecución para todos los usuarios.

Con estas herramientas en las manos, ya es posible escribir un conjunto de comandos dentro de un archivo de texto. Para que el Shell interprete ese archivo como un ejecutable, será necesario asignarle permisos de ejecución.

Para ejecutar el script en un entorno Shell cualquiera, se puede proceder de dos formas:

* shell <nombre del script>. En este caso, se cargará el Shell que se especifique: bash, sh, ksh, etc., al cual se le pasará, como argumento, el nombre del programa que se desea ejecutar, por ejemplo:
  - bash Example. En esta orden, se le pide al Bash que ejecute el archivo Example.
* ./<nombre del script>. El símbolo ./ se refiere al directorio actual, es decir, se le dice al Shell que el archivo que tiene que ejecutar está en el directorio actual, por ejemplo:
  - ./Example. Obsérvese que la cadena ./Example especifica un camino y por eso no puede contener espacios.

El primer programa que se presenta (V.1) es bien sencillo y lo único que contendrá son tres comandos: clear para limpiar la pantalla, pwd (*print working directory*) para conocer el nombre del directorio de trabajo actual y echo para imprimir algo. La opción -n del comando echo inhibe el cambio de línea que escribe el comando por defecto.

El programa V.1 también contiene un comentario: el símbolo # hace que el intérprete ignore todos los caracteres que le siguen hasta el cambio de línea.

```
#Programa V.1. Ejemplo elemental de Shell script
clear
echo Este es un programa en Shell script
echo -n El directorio actual o de trabajo es:
pwd
```

La salida del programa será (el directorio variará de acuerdo con el directorio actual):

```
Este es un programa en Shell script
El directorio actual o de trabajo es: /home/mlezcano
```

Obsérvese que el último comando echo y el comando pwd imprimen sobre la misma línea, debido a que el segundo echo usa la opción -n para inhibir el cambio de línea que siempre da por defecto este comando.

Los programas en Shell script pueden contener una primera línea que especifica el Shell que ejecutará el archivo script; si no se especifica nada, como en el programa V.1, el Shell es el que esté usando el usuario que ordenó la ejecución.

La variable de ambiente **SHELL** contiene el nombre del Shell que se está usando. Para ver su contenido, se puede usar el comando echo $SHELL que imprimirá /bin/bash si se está usando el Bash (no tiene que ser ese directorio específico). Recuerde que el signo $ no forma parte del nombre de la variable y se usa cuando se quiere acceder a su contenido.

```
#!/bin/bash
#Programa V.1. Ejemplo elemental de Shell script (modificado)
clear
echo Este es un programa en Shell script
echo -n El directorio actual o de trabajo es:
pwd
```

La segunda versión del programa V.1 muestra una primera línea especial que comienza con #![58], a fin de especificar que el programa lo debe ejecutar el Bash que está localizado en el camino o ruta /bin/, es decir, en el directorio bin que es hijo del directorio raíz (/).

Es una práctica muy buena poner siempre, en los programas Shell script, esa primera línea; si no se hace así, se usará el intérprete por defecto y las órdenes contenidas en el programa pueden ser incompatibles con él. Este tipo de línea es común en los lenguajes interpretados, tales como Perl, Phyton, etc.

El programa V.2 amplía el programa V.1 poniendo las cadenas que se asocian a los comandos echo entre comillas (") y agregándoles otras funcionalidades. Cuando el comando echo se encuentra una cadena entre comillas, interpreta los caracteres especiales[59].

```
#!/bin/bash
#Programa V.2. Trabajando con variables de usuario, de ambiente y posicionales

clear
today=`date`
set $today
echo "Hola $USER, hoy es $1 $3-$2-$6. Son las $4"
exit 0
```

También se usan tres nuevos comandos: date, que permite escribir la fecha; set, que fija valores del Shell y de los parámetros posicionales; y exit[60], que termina el proceso.

58 La combinación de símbolos #! al inicio del programa se conoce como *sha-bang*, y es una marca especial que se usa en los sistemas tipo Unix para indicar el camino absoluto donde se encuentra el programa que debe interpretar al script contenido en el archivo.

59 Un carácter especial es aquel que debe interpretarse de manera diferente a su significado literal.

60 Todos los comandos con comportamiento correcto, en sistemas tipo Unix, devuelven un estado de salida (representado por un número entero): si tiene éxito, el retorno es 0 y si no tiene éxito, es distinto de cero.

La salida del programa V.2 será algo como lo siguiente:

```
Hola mlezcano, hoy es Mon 7-Sep-2016. Son las 17:37:18
```

Se deben hacer algunas aclaraciones en relación con el programa V.2:

1. Se usan tres tipos de variables: de usuario (today), de ambiente (USER) y posicionales (representadas por un número). Estos aspectos se trataron en el capítulo IV.
2. La orden today=`date` es una sentencia de asignación (el signo = es el operador de asignación); en esta sentencia, se le asigna a la variable today un cierto valor. No pueden dejarse espacios entre la variable que recibe el valor, el operador de asignación y el valor que se asignará.
3. Cuando el Shell encuentra una orden del tipo `comando`, es decir, encerrada entre apóstrofos invertidos, a la derecha de una sentencia de asignación, la interpreta ejecutando el comando. Por eso, en la sentencia today=`date` la variable today recibe el resultado de ejecutar el comando date, que es una cadena de valores separados por espacios en la forma: Mon Sep 7 17:37:18 EDT 2016.
4. El comando set $today hace que las variables posicionales, desde la 1 hasta la que se necesite, tomen los valores de los campos de la variable today (obsérvese que para referirse a una variable es necesario antecederla con el signo $, pero ese signo no forma parte de su nombre).

   En este caso, serían seis campos porque hay seis cadenas de caracteres separadas por espacios, de modo que las variables posicionales ($1, $2, $3, $4, $5 y $6) quedan con los valores siguientes:

   - $1 toma el valor Mon, es decir, el día de la semana.
   - $2 recibe el valor Sep, que es el mes actual.
   - $3 toma el valor del numeral del día dentro del mes (7).
   - $4 contiene la hora en ese momento (17:37:18).
   - $5 se fija a la zona horaria en uso (EDT).
   - $6 almacena el año (2016).
5. Cuando el comando echo se encuentra la referencia a una variable (tiene la forma $Var) entre comillas, interpreta que debe imprimir el contenido de esa variable. Por eso, cuando se encuentra $USER, imprime el contenido de la variable de ambiente USER, que es el nombre con el que el usuario entró al sistema, y cuando se encuentra $n, imprime el valor de la variable posicional correspondiente n (n es un entero).
6. La sentencia exit 0 al final del programa le asigna el código de retorno 0 al programa; ese código se toma (por convenio) como el aviso de que no hubo errores. Un código diferente de cero significa que hubo algún tipo de error. El código de retorno de los programas se almacena en la variable especial ?.

Este pequeño programa, aunque simple, tiene muchos detalles que están presentes en la programación en Shell script, debe tratar de entenderlo en su totalidad.

V.3 PROGRAMACIÓN EN SHELL SCRIPT

Los ejemplos presentados en la sección anterior ofrecen una idea general acerca de la programación en Shell script. Esta sección también comienza con un ejemplo (el programa V.3), que tiene la intención de insistir acerca de las variables dentro de un programa Shell script.

El programa V.3 comienza imprimiendo algo, para lo cual usa el comando:

```
echo "El programa $0, lo ejecuta $USER en la computadora $HOSTNAME."
```

```
#!/bin/bash
#Programa V.3
echo "El programa $0, lo ejecuta $USER en la computadora $HOSTNAME."   #1
echo "La terminal que se usa es de tipo $TERM."                        #2
echo "El directorio de trabajo es $PWD."                               #3
echo "El directorio de inicio de $USER es $HOME."                      #4
echo "Se usa el conjunto de caracteres $LANG."                         #5
echo "El programa recibe los argumentos: $1, $2 y $3."                 #6
exit 0
```

Debe observarse que es obligatorio que la cadena que le sigue al comando echo esté entre comillas, debido a que así el Shell interpreta algunos símbolos de forma especial. En este caso particular, interpretará que debe imprimir el contenido de las variables: 0, USER y HOSTNAME. Si la cadena estuviera entre apóstrofos, por ejemplo, 'USER' o solo USER, echo la imprimiría literalmente, es decir, sin hacer ninguna interpretación; en este caso imprimiría la palabra USER.

La variable posicional 0 contiene el nombre del programa, y las variables de ambiente USER y HOSTNAME contienen el identificador del usuario y el nombre de la máquina, respectivamente; así que ante la línea que se está analizando (con el comentario #1 en el programa), el Shell imprimirá lo siguiente (suponiendo que el programa se llama progv3, el usuario es mlezcano y la máquina es miDebian):

```
El programa progv3, lo ejecuta mlezcano en la computadora miDebian.
```

Los siguientes comandos echo (observe los números puestos como comentarios para facilitar la explicación) imprimen los contenidos de otras variables:

* Línea #2. El tipo de terminal, contenido en la variable de ambiente TERM.
* Línea #3. El directorio de trabajo, almacenado en la variable de ambiente PWD.
* Línea #4. El identificador del usuario, que está almacenado en la variable de ambiente USER, y su directorio de inicio, que se almacena en la variable de ambiente HOME.

* Línea #5. El conjunto de caracteres usado, almacenado en la variable de ambiente LANG.
* Línea #6. Tres variables posicionales (1, 2 y 3): 1 - primer parámetro, 2 - segundo parámetro y 3 - tercer parámetro. Observe que el programa no hace nada para forzar a que se pasen esos parámetros, esta situación se resolverá más adelante; por eso, si no se le pasan los parámetros al programa, esas variables estarán vacías.

V.3.1 Operadores

V.3.1.1 La sentencia test y sus operadores

El lenguaje Shell script posee un conjunto de operadores que se usan para evaluar expresiones; ellos actúan sobre varios tipos de datos. En esta sección, se presentan dichos operadores clasificados según diferentes criterios.

La sentencia test se usa para evaluar expresiones y tiene dos formas que son semánticamente equivalentes:

* test <expression>
* [<expression>]

El nombre de ambas formas es test y son equivalentes aunque sintácticamente se escriban de forma diferente. Cuando se usa la segunda forma, el comando es en realidad el corchete abierto ([), aunque siempre habrá que cerrarlo para dar fin a la expresión que se desea evaluar. Para pedir ayuda en el manual acerca de esta sentencia, se puede usar cualquiera de estas dos formas:

* man test
* man [

Las expresiones dentro de test se pueden combinar usando diversos operadores, las tablas que se presentan a continuación los agrupan según diferentes criterios.

La tabla V.1 muestra un grupo de operadores que se usan para realizar comparaciones dentro de la sentencia test. Están agrupados de acuerdo con el contenido de las variables.

* Antes de ver las tablas, es importante hacer algunas aclaraciones importantes:
* Ante una expresión en la forma [$a=$b], el Shell reportará error sintáctico porque deben existir espacios entre cada una de las partes de la expresión; la forma correcta de la expresión anterior es: [$a = $b].
* La forma test $a=$b no reporta error, pero el resultado no será el esperado y por eso debe escribirse como test $a = $b.
* La sentencia test devuelve 0 si la comparación es verdadera y 1 si es falsa (obsérvese que no es lo mismo devolver que imprimir).

Tabla V.1.

Operadores básicos de test

SUPÓNGASE QUE PREVIAMENTE SE HAN HECHO LAS OPERACIONES: a=7; b=9			
OPERADOR	**SIGNIFICADO**	**EJEMPLO**	**COMENTARIO**
-a	Verdadero si sus dos operandos también son verdaderos	[$a -a $b]	El valor devuelto es 0, es decir, verdadero, porque a y b son verdaderos. Una variable con contenido es verdadera y sin contenido es falsa.
!	Niega la expresión	[! $a]	El valor devuelto es 1, es decir, falso, porque $a es verdadero y se niega la expresión.
-o	Verdadero si uno de sus operandos es verdadero	[$a -o ! $b]	El valor devuelto es 0, es decir, verdadero, porque a es verdadero.

El valor devuelto al evaluar cualquier expresión en el Bash se asigna a la variable especial ?; se puede imprimir su contenido haciendo uso de la sentencia echo $?

La tabla V.2 muestra los operadores relacionales que se usan para comparar números enteros (todos son infijos[61] y binarios[62]).

Tabla V.2.

Operadores relacionales numéricos de test. Los operandos son números enteros

SUPÓNGASE QUE PREVIAMENTE SE HAN HECHO LAS OPERACIONES: a=3; b=4			
OPERADOR	**SIGNIFICADO**	**EJEMPLO**	**COMENTARIO**
-eq (equal)	Igual que	[$a -eq $b]	El valor devuelto es 1, es decir, falso: 3 no es igual que 4
-ge (greater o equal)	Mayor o igual que	[$b -ge $a]	El valor devuelto es 0, es decir, verdadero: 4 es mayor o igual que 3
-gt (greater than)	Mayor que	[$a -gt $b]	El valor devuelto es 1, es decir, falso: 3 no es mayor que 4
-le (less or equal)	Menor o igual que	[$a -le $b]	El valor devuelto es 0, es decir, verdadero: 3 es menor o igual que 4
-lt (less than)	Menor que	[$b -lt $a]	El valor devuelto es 1, es decir, falso: 4 no es menor que 3
-ne (not equal)	No igual que	[$a -ne $b] [$b -ne $a]	El valor devuelto es 0, es decir, verdadero: 3 y 4 son desiguales

Ejercicio:

Desde una terminal, teclee las expresiones que se muestran en la columna "Ejemplo" de las tablas V.1 y V.2, después ejecute el comando echo $?. Observe que el valor que se imprimirá dependerá del valor de retorno de cada expresión evaluada por la sentencia test. Por ejemplo, teclee: a=3; b=4; [$a -eq $b]; echo $?, ¿qué valor se imprime?, ¿por qué?

61 Operadores infijos, son aquellos que están dentro de dos operandos: <operando1>operador<operando2>.

62 Operadores binarios, son aquellos que tienen dos operandos.

La tabla V.3 muestra que los operadores para el trabajo con cadenas alfanuméricas son de dos tipos: unarios[63] y binarios. La tabla V.4 muestra los operadores lógicos que se pueden usar con la sentencia test.

Después de analizar las tablas V.3 y V.4, realice los ejercicios propuestos. Debe asegurarse de comprenderlos. La idea que usted siempre debe seguir, en esta etapa del aprendizaje, es comprobar los comandos desde el *prompt* del intérprete de comandos para que después pueda incluirlos en los programas Shell script que haga.

Tabla V.3.

Operadores para comparar cadenas alfanuméricas con test

SUPÓNGASE QUE PREVIAMENTE SE HA HECHO LA OPERACIÓN: s1=MERCEDES; s2=DANIEL			
OPERADORES UNARIOS			
OPERADOR	**SIGNIFICADO**	**EJEMPLO**	**COMENTARIO**
-n	La longitud de la cadena es distinta de cero	[-n $s1]	El valor devuelto es 0, es decir, verdadero. La longitud de la cadena contenida en la variable s1 no es cero
-z	La longitud de la cadena es cero	[-z $s3]	El valor devuelto es 0, es decir, verdadero. La longitud de una variable que no ha tomado valor es cero
OPERADORES BINARIOS			
=	Igual que	[$s1 = $s2]	El valor devuelto es 1, es decir, falso. Las variables $s1 y $s2 contienen cadenas diferentes.
!=	Desigual que	[$s1 != $s2]	El valor devuelto es 0, es decir, verdadero. Las cadenas contenidas en las variables son diferentes

Tabla V.4.

Operadores lógicos que usa la sentencia test

SUPÓNGASE QUE PREVIAMENTE SE HA HECHO LA OPERACIÓN: s1=MARIEN; s2=LAUREN; s3=LISA			
OPERADOR UNARIO			
OPERADOR	**SIGNIFICADO**	**EJEMPLO**	**COMENTARIO**
!	Negación	! [-z $s1]	El valor devuelto es 0, es decir, verdadero, porque la expresión [-z $s1] es falsa y su negación es verdadera
OPERADORES BINARIOS			
&&	AND, solo es verdadera si ambos operandos son verdaderos	[$s1 = $s2] && [$s1 != $s3]	El valor devuelto es 1, es decir, falso, porque la expresión [$s1 = $s2] es falsa
||	OR, solo es falsa si todos los operandos son falsos	[$s1 = $s2] || [$s1 != $s3]	El valor devuelto es 0, es decir, verdadero, porque la expresión [$s1 != $s3] es verdadera

63 Operadores unarios, son aquellos que tienen un solo operando.

Ejercicios:

1. Use los operadores de las tablas V.3 y V.4 desde una terminal. Proceda de la siguiente manera:
   a. Asigne valores a algunas variables de usuario. Observe cómo hacerlo en la segunda fila de la tabla V.3.
   b. Pruebe cada uno de los ejemplos mostrados en la tabla V.3, haciendo las comparaciones que se muestran en la columna correspondiente.
      i. Para cada comparación que haga, averigüe el valor de la variable especial ? (contiene el valor devuelto por el último comando).
2. Pruebe los ejemplos mostrados en la tabla V.4.
3. Proponga ejercicios de este tipo usted mismo y pruebe que funcionen en la terminal, después intente usar esos resultados dentro de un programa en Shell script.

Sugerencia: Use comandos compuestos, por ejemplo: s1=Daniel; test -n $1; echo $?

La tabla V.5 muestra los operadores que se usan junto a la sentencia test para obtener diversas informaciones acerca de los archivos.

Estos operadores son muy importantes cuando se programa en Shell script, debido a que muchas veces los programas se hacen para interaccionar con archivos y es necesario conocer algunas de sus características, tales como:

* ¿El archivo existe?
* ¿Cuáles son los permisos que se tienen sobre el archivo? (Escritura, ejecución, lectura).
* ¿Qué tipo de archivo es? (Un directorio, un archivo especial, un archivo ordinario, un socket, un descriptor de archivo, una tubería nombrada, etc.).
* ¿La longitud del archivo es diferente de cero?
* ¿El archivo tiene fijado el SGID?
* ¿Los archivos son iguales?, ¿tienen la misma fecha?

Es habitual usar la sentencia test para probar diversas condiciones acerca de los archivos y después tomar alguna alternativa basada en las características del archivo.

Analice la tabla V.5 de manera detallada y asegúrese de comprender todos los aspectos que se presentan en ella. Observe que son diversos y tendrá que cerciorarse de que los entiende. Después, puede proceder a realizar los ejercicios que se proponen. Hágalo en las dos formas propuestas, es decir, interactivamente y dentro de un programa en Shell script.

Tabla V.5.

Operadores que actúan sobre archivos

OPERADORES UNARIOS			
OPERADOR	**SIGNIFICADO**	**EJEMPLO**	**SIGNIFICADO**
-b (*block*)	Archivo de bloques	-b file1	Verdadero si existe el archivo especial de bloques file1
-c (*character*)	Archivo de caracteres	-c file2	Verdadero si existe el archivo especial de caracteres file2
-d (*directory*)	Es un directorio	-d dir	Verdadero si existe el directorio dir
-e (*exist*)	El archivo existe	-e file3	Verdadero si existe el archivo file3 (aunque sea un directorio)
-f (*file*)	Archivo regular	-f file4	Verdadero si existe el archivo regular file4
-G	Efectivo ID de grupo	-G file5	Verdadero si existe el archivo file5 y pertenece al *effective group* ID
-g (*group*)	Tiene fijado el SGID	-g file6	Verdadero si existe el archivo file6 y tiene fijado su bit SGID
-h	Enlace simbólico	-h file7	Verdadero si el archivo file7 es un enlace simbólico
-k	Sticky bit	-k file8	Verdadero si existe el archivo file8 y tiene fijado su *sticky bit*
-L	Es un enlace simbólico	-L file9	Verdadero si el archivo file9 es un enlace simbólico
-o	Efectivo ID de usuario	-o file10	Verdadero si existe el archivo file10 y pertenece al *effective user* ID
-p	Tubería con nombre	-p file11	Verdadero si el archivo file11 es una tubería con nombre
-r	Lectura (*read*)	-r file12	Verdadero si el archivo file12 tiene permiso de lectura
-s	Longitud mayor que cero	-s file13	Verdadero si la longitud del archivo file13 es mayor que cero
-S	Socket	-S file14	Verdadero si existe el archivo file14 y es un socket
-t	Terminal	-t fileD	Verdadero si fileD es un descriptor de archivo que refiere a una terminal
-u	*Bit SUID*	-u file15	Verdadero si el archivo file15 tiene fijado su bit *SUID*
-w	Escritura (*write*)	-w file16	Verdadero si el archivo file16 tiene permiso de escritura
-x	Ejecución (*execute*)	-x file17	Verdadero si el archivo file17 tiene permiso de ejecución
OPERADORES BINARIOS			
-ef (*equal files*)	Son archivos iguales	file1 -ef file2	¿Los archivos file1 y file2 tienen el mismo i-node y están en el mismo equipo (apuntado desde lugares distintos)?
-nt (*newer than*)	Más reciente que	file1 -nt file2	¿El archivo file1 es más reciente que el file2?
-ot (*older than*)	Más viejo que	file1 -ot file2	¿El archivo file1 es más viejo que el archivo file2?

Ejercicios:

1. Compruebe, de forma interactiva, algunas de las características de los archivos que están en su directorio de trabajo. Recuerde combinar test con echo $? para conocer el resultado de la evaluación que ha hecho test; por ejemplo, compruebe lo siguiente: [-e file]; echo $?
2. Si el archivo file existe, test fijará la variable ? con el valor 0 y si no existe, le asignará el valor 1. Con base en eso, echo $? imprimirá un valor o el otro.
   Ahora intente hacer ese mismo tipo de prueba con los ejemplos mostrados en la tabla V.5. Recuerde que puede usar el comando chmod para cambiar algunos permisos de los archivos.
3. Vaya más adelante y lleve estas mismas ideas a un programa.

V.3.1.4 Operadores aritméticos

La tabla V.6 muestra las operaciones aritméticas básicas que se pueden hacer en el Bash y en muchos otros lenguajes script.

Tabla V.6.

Operadores aritméticos básicos

OPERADOR	SIGNIFICADO	EJEMPLO	RESULTADO
+	Suma	3 + 4	7
-	Resta	5 - 2	3
*	Multiplicación	12 * 3	36
/	División	18 / 6	3
%	Resto de la división entera	19 % 6	1

En el Shell, no existen tipos de variables y por eso una misma variable en un momento puede contener una palabra o un número. Es responsabilidad del programador pasar valores numéricos a estas expresiones.

Existen cuatro formas de evaluar una expresión aritmética:

La primera forma es usando el comando externo expr. A continuación, se muestran algunos ejemplos. El comando expr imprime el valor calculado:

```
a=11; b=5;
expr $a + $b          #Imprime el resultado de sumar a y b.
c= `expr $a - $b`     #Resta el valor b de a y le asigna el resultado a c.
expr $a \* $b         #Multiplica a por b e imprime el resultado. Obsérvese que es
                      #necesario usar el carácter de escape (\).
c=`expr $a / $b`      #divide a entre b y asigna el resultado a c.
expr $a % $b          #Halla el resto de la división entera de a entre b e imprime el resultado.
```

El comando expr, heredado del Bourne shell, es un comando externo y por eso la operación se hace más lenta.

Es necesario dejar los espacios antes y después de los operadores o la expresión no se evaluará, por ejemplo: expr $a+$b imprimirá 11+5, es decir que tomará los valores de las variables y los imprimirá junto con el operador sin evaluar la expresión.

* La segunda forma es usando el comando interno let, el cual evalúa la expresión pero no imprime el resultado, como sucede con expr cuando se usa de forma independiente, como se hizo en expr $a + $b y en expr $a % $b.
  Ejemplos:

```
d=34; f=12;
let h=$d+$f          #Asigna a la variable h la suma de las variables d y f.
let h=d+f            #Es una expresión equivalente a la anterior
let h=d-f            #Resta el valor f de d y lo asigna a la variable h.
let h= d*f           #Multiplica d por f, obsérvese que no es necesario
                     #usar el carácter de escape (\).
let h=d/f            #Divide d entre f y lo asigna a h.
let h=d%f            #Le asigna a h el resto de la división entera de d entre f.
```

El comando let informa error sintáctico si se usan espacios antes o después de los operadores; al contrario de expr, en el que es necesario dejar espacios a ambos lados de los operadores.

Cuando se usa el comando let, no es necesario que las variables estén antecedidas por el signo $ para referirse a su contenido (aunque se puede usar).

* La tercera forma de evaluar expresiones en el Bash es poniéndolas entre doble paréntesis. En este caso, existe flexibilidad con los espacios que se pueden dejar o no, según considere el usuario, y en el uso del signo $ precediendo las variables.
  Ejemplos:

```
m=23; n=10;
$((m+n))             #Suma las variables m y n.
f=$(($m -$n))        #Resta el valor n de m y lo asigna a f.
$(($m * $n))         #Multiplica m por n.
c=$(($m / $n))       #Divide m entre n, y asigna el resultado a c.
$(($m % $n))         #Halla el resto de la división entera de m entre n.
```

Tanto el comando let, como la forma de doble paréntesis admiten tres operadores adicionales: incremento (++), decremento (--) y exponenciación (**). Los dos primeros operadores son unarios y el último es binario. Observe los ejemplos siguientes en los cuales se mezclan ambas formas:

```
a=2; b=4;
let b**a             #Eleva b a la a.
$((b ** a))          #Efectúa la misma operación que la anterior
(( b++ ))            #Suma 1 al valor de b.
let b++              #Efectúa la misma operación que la anterior
(( --b ))            #Resta 1 al valor de b.
let --b              #Efectúa la misma operación que la anterior
```

Los operadores ++ y -- pueden ser prefijos (van antes del operando) o posfijos (van después del operando).

Todas las operaciones aritméticas se realizan con números enteros. Si se desea hacer una operación con números en punto flotante, es necesario usar la calculadora bc, que incluye un lenguaje de programación.

V.3.2 Sentencias condicionales

Las sentencias condicionales, de cualquier lenguaje de programación, permiten tomar decisiones para dirigir el flujo de ejecución de un programa hacia algún lugar específico.

Sentencia if

A continuación, se formaliza la definición de la sentencia condicional if, que se toma como ejemplo para establecer las normas sintácticas generales que se usan en este capítulo.

Sintaxis de if:

if $clist_1$; then $clist_2$; [[$elif_1$ $celist_1$; then $celist_2$;] ... [$elif_n$ $celist_n$; then $celist_n$;]][else list;] fi

En las definiciones sintácticas usadas en este libro, cualquier expresión entre corchetes ([]) es opcional, de modo que la forma más simple de la sentencia if es la que se usa en el programa V.4, es decir: if list1; then list2; fi.

El programa V.4 comienza con una sentencia if que debe evaluar una expresión usando la sentencia test en su forma []. Observe que en el programa V.4 hay un espacio entre los corchetes (abiertos y cerrados), los cuales deberán estar siempre; de lo contrario la forma [] no se reconocerá sintácticamente por el Shell.

```
#!/bin/bash
#Programa V.4

if [ $# -ne 3 ]
  then
    echo "Error. Debe pasar tres argumentos"
    exit 1
fi

echo "El programa $0, lo ejecuta $USER en la computadora $HOSTNAME."
echo "La terminal que se usa es de tipo $TERM."
echo "El directorio de trabajo es $PWD."
echo "El directorio de inicio de $USER es $HOME."
echo "Se usa el conjunto de caracteres $LANG."
echo "El programa recibe los argumentos: $1, $2 y $3."
exit 0
```

Una expresión de la forma [<operation>], como la usada en el programa V.4, puede sustituirse por otra expresión en la forma test <operation>. De modo que el if del programa anterior puede reescribirse de la manera siguiente: if test $# -ne 3.

La comparación a efectuar en el programa V.4 es: $# -ne 3, en donde -ne es el operador binario de desigualdad (***not equal***). En este caso, se compara el valor de la variable especial # (cantidad de argumentos pasados al programa) con 3; si la cantidad de argumentos no es correcta, el programa emite un mensaje de error y termina usando la sentencia exit con un código de retorno diferente de cero, lo cual significa que hubo algún tipo de error.

Obsérvese que en la definición sintáctica del if se usa el punto y coma (;) como separador de los lexemas[64] que forman parte de la sentencia[65]. En el ejemplo V.4, esta variante se escribió en la forma:

```
if [ $# -ne 3 ]
  then                                  # El then debe sangrarse un espacio, él forma parte del if.
    echo "Error. Debe pasar tres argumentos"  # Las acciones correspondientes al then se sangran
    exit 1                              #todas al mismo nivel.
fi                                      #El fin del if (fi) está en línea con su inicio (if).
```

En este caso, no fue necesario el uso de punto y comas para separar los lexemas, debido a que los cambios de línea hacen la función de separadores de lexemas y el Shell puede discriminar entre cada uno de ellos. Se recomienda escribir las sentencias de los programas de esta manera, es decir, separando los lexemas por cambios de líneas; también deben identarse[66] las sentencias debido a que resulta más fácil leer el código. Observe los comentarios del segmento de código anterior.

```
$if ls; then pwd; fi
f1 f2 f3
/home/dlezcano/prueba
$if  ls
>then
>pwd
>fi
f1 f2 f3
/home/dlezcano/prueba
$
```

Figura V.3. Ejecución del comando if en forma interactiva.

Como ya se ha dicho, lo que se ha denominado sentencia en este libro, para usar palabras propias de los lenguajes de programación, es en realidad un comando (o subcomando) y por eso se pueden ejecutar desde el *prompt* del SO. Observe la figura V.3 (recuerde que lo que se escribe en negrita es la salida del SO).

64 Lexemas. Unidadas lógicas generadas durante la interpretación o compilación de un programa.

65 Muchos textos y el manual de ayuda de Unix (man) se refieren a este tipo de facilidades (if, for, etc.) como comandos o subcomandos. En este texto, se ha preferido llamarlas sentencias por su similitud con esas construcciones de los lenguajes de programación y de acuerdo con la notación seguida en otros libros.

66 Identar (sangrar). Es un anglicismo, común en informática, que significa mover un bloque de texto hacia la derecha insertando espacios para separarlo del margen izquierdo y distinguir el alcance del bloque.

* En la primera forma: if ls; then pwd; fi, se ponen los punto y coma para separar las partes (lexemas) que conforman la sentencia if.
* En la segunda forma, se escriben los lexemas en líneas separadas. En este caso, cada vez que se oprime la tecla enter (cambio de línea), el Shell se percata de que el comando no está completo y por eso lanza su segundo *prompt* (por defecto el signo >, que está contenido en la variable de ambiente PS2), con lo que indica que faltan elementos. Cuando el Shell se encuentra con el lexema fi, identifica el final de la sentencia que le indica que ya puede intentar reconocerla.

Obsérvese que la lista de comandos del if, en la figura V.3, está compuesta por el comando ls que nunca falla (su código de retorno es 0), y por eso se pasa a ejecutar el comando que está en la parte then (pwd).

Se retoma la sintaxis de la sentencia if para interpretar su significado (su semántica):

if $clist_1$; then $clist_2$; [[$elif_1$ $celist_1$; then $celist_2$;] ... [$elif_n$ $celist_n$; then $celist_n$;]][else list;] fi

La interpretación de la forma sintáctica anterior es la siguiente:

1. Se ejecuta la parte if de la sentencia ($clist_1$ en este caso). Si su valor de retorno es cero (es verdadero), entonces se ejecuta la parte then ($clist_2$) y termina.
2. Si $clist_1$ tiene valor de retorno distinto de cero (es falso) y existe la parte elif (es opcional), se ejecuta cada $elif_i$ (i = 1,..., n) hasta que se encuentre algún $elist_i$ que sea verdadero; en ese caso, se ejecuta su parte then (then $celist_i$) y termina.
3. Si $clist_1$ y todos los $elif_i$ son falsos, se ejecuta la parte else.

El programa V.5 muestra un ejemplo del uso de la sentencia if con todas sus partes. Debe observarse que se utiliza una sentencia if que contiene dos elif y el else, es decir que este programa solo contiene una sentencia. En la parte if y en todas las partes elif, se tiene que probar alguna condición, no así en la parte else a la cual se llega si fallan todas las pruebas anteriores.

Primero se verifica que se ha pasado la cantidad de parámetros correcta (2), lo cual se hace evaluando la expresión ! [$# -eq 2]; obsérvese que se niega (!) la expresión de la sentencia test ([]), es decir que si la cantidad de parámetros no es 2, entonces la evaluación de la sentencia [$# -eq 2] es falsa y por tanto su negación es verdadera, lo que significa que hay un error (los espacios que se dejan son de carácter obligatorio). En ese caso (hubo error), se ejecuta la parte then, donde se imprimen las cadenas asociadas a la sentencias echo y el programa termina con código de error (exit 1).

Si la cantidad de parámetros es correcta, es decir, si la evaluación de la expresión que acompaña al if es falsa, se pasa a evaluar el primer elif, para comprobar si el primer parámetro (contenido en la variable posicional 1) es el nombre de un directorio, de nuevo se usa la expresión negada; si es verdadera, el programa ejecuta la parte then correspondiente a ese elif y termina informando un error (exit 2).

```
#!/bin/bash
#Programa V.5. Encuentra los archivos de acuerdo al tipo

if ! [ $# -eq 2 ]
  then
    echo "Sintaxis incorrecta". Debe invocar al programa en la forma siguiente:"
    echo "$0 <Nombre de directorio> <tipo de archivo>"
    echo " el tipo de archivo puede ser:"
    echo "  f (ordinario), d (directorio), l (enlace)"
    exit 1
  elif   ! [ -d $1 ]
   then
    echo "$1 no es un directorio"
    exit 2
  elif [ $2 = "d" ] || [ $2 = "f" ] || [ $2 = "l" ]
    then
      find $1 -maxdepth  1 -type $2
      exit 0
 else
    echo "Opción no válida"
    exit 3
fi //Fin del primer if y del programa
```

El segundo elif comprueba si la opción es válida, para lo cual se usa el operador lógico or (||). Si la variable posicional 2 contiene el carácter d, f o l, será verdadera la expresión siguiente: [$2 = "d"] || [$2 = "f"] || [$2 = "l"].

En ese caso, se ejecuta la parte then del elif, en la cual se ejecuta el comando find, que usa los valores contenidos en las variables posicionales 1 y 2.

Por último, si el programa llega a la parte else, es debido a que todas las pruebas efectuadas en el if y en los elif que le preceden fueron falsas y se puede afirmar que el error fue en la opción.

Sentencia case

La sentencia case también es una condicional que permite dirigir el flujo de un programa en ejecución hacia diferentes partes del código. Cuando existen muchas alternativas de ejecución es mejor usar esta sentencia y no un if, con muchas partes elif, debido a que el programa es más fácil de comprender y queda más compacto.

La sintaxis de la sentencia case es la siguiente:

```
case word in [ [(] pattern [ | pattern ] ... ) list ;; ] ... esac
```

Debe recordar que en la forma sintáctica adoptada, las expresiones entre corchetes son opcionales, por eso la forma más sencilla de case es: case word in esac.

El programa V.6 consta de un menú con tres opciones que se muestran usando el comando echo, después de lo cual se usa el comando read A, que lee la selección del usuario y la almacena en la variable A.

```
#!/bin/bash
# Programa V.6

echo "Puede obtener información acerca de:
 1. Datos acerca de la ejecución del sistema.
 2. La utilización del disco.
 3. Los procesos en ejecución."
echo -n "Escoja su selección numérica (de 1 a 3): "

read A
clear

case $A in
  1)
    echo "El nombre del host es: $HOSTNAME."
    dat=`uptime`
    set $dat
    echo "La hora actual es: $1."
    echo "El sistema se ha estado ejecutando por $3 $4."
    echo "hay $5 usuarios conectados."
    echo "El promedio de carga del sistema es:"
    echo " $9 en el último minuto, $10 en los últimos 5 minutos"
    echo -n " y $11 en los últimos 15 minutos."
    echo;;
  2)
    echo "Reporte de espacio en disco"
    df –h;;
  3)
    echo "Procesos en ejecución"
    ps;;
  *)
    echo "Entrada no válida"
    exit 1;;
esac  //Fin de la sentencia case y del programa
```

La sentencia case que se usa tiene la siguiente forma general:

```
case $A in     #No se ponen los detalles que deberán verse en el programa
  1) ;;
  2) ;;
  3) ;;
  *)
    echo “Entrada no válida”
    exit 1;;
esac
```

Observe que hay una etiqueta por cada uno de los valores válidos que puede tomar la variable (A, en este caso). En este ejemplo, las etiquetas son: 1), 2) y 3); además, hay una etiqueta especial, *), para la acción por defecto.

En el caso del programa V.6, la acción por defecto es emitir un mensaje de error que le informa al usuario que ha elegido una opción que no es correcta (algo distinto a los valores: 1, 2 o 3), para después salir con el código de error 1 (exit 1).

Cuando un programa se encuentra ante una sentencia con la forma case word in, que en el ejemplo es case $A in, compara el valor de la variable con cada una de las etiquetas y salta a la etiqueta que coincide con el valor de la variable (A, en este caso) para ejecutar todas las instrucciones desde ese lugar hasta el fin del alcance de la etiqueta que va hasta los dos símbolos de punto y coma (;;). Si la selección del usuario no coincide con ninguna de las etiquetas, es decir: 1), 2), 3), el salto se produce a la etiqueta *).

En el ejemplo V.6, el programa, según la opción escogida, mostrará información acerca de:

* Datos de la ejecución del sistema.
* Datos de la utilización del disco.
* El listado de los procesos en ejecución.

V.3.3 Sentencias de repetición

Los lenguajes de programación permiten repetir un grupo de sentencias las veces que sean necesarias; el lenguaje Shell script del intérprete Bash también posee esas facilidades. En particular, existen cuatro sentencias de repetición o ciclos: for, while, until y select, que se verán a continuación.

Sentencia for

La sentencia for tiene dos formas sintácticas.

Primera forma sintáctica de la sentencia for

```
for name [ [ in [ word ... ] ] ; ] do command; done
```

El funcionamiento de la sentencia es el siguiente: la lista de palabras ⌈ word ... ⌉ se expande, generando una lista de ítems que se toman como valores de la variable name (uno a uno). Para cada ítem, se ejecuta command. Si la parte in word se omite, for ejecuta command para cada uno de los parámetros posicionales que se fijen. El estado de salida de for es el estado del último comando que se ejecute; si no se expanden ítems, no se ejecuta ningún comando y el estado de salida es cero.

El programa V.7 muestra un ejemplo del uso de esta forma de la sentencia for. Lo primero que hace el programa es usar la sentencia test para verificar si el programa recibió argumentos.

* Si el programa no recibe argumentos, es decir, si el valor de la variable especial # es igual que cero, se le asigna a la variable Files el listado del directorio actual (solo los nombres).
* Si se le pasan argumentos al programa, se asignan a la variable Files y se supone que son nombres de archivos. Los argumentos que se pasan a un programa quedan contenidos en la variable especial *.

De esta manera, la variable Files contiene una lista con nombres de archivos. Obsérvese que en el segundo caso no se verifica si son en realidad nombres de archivos.

Después, se hace un ciclo for tantas veces como nombres haya en la variable Files; en cada iteración, se verifica si la variable file contiene el nombre de un archivo y se imprime su contenido (usando el comando cat).

```
#!/bin/bash
# Programa V.7

if [ $# -eq 0 ]
  then
    Files=$(ls)
  else
    Files="$*"
fi

for file in $Files
  do
    if [ -d "$file" ]
      then
        echo "$file es un directorio no se toma en cuenta"
      else
        if [ -e $file ]
              then
                echo "Contenido del archivo $file:"; cat $file
        fi
    fi
  done  //Fin del for y del programa
```

Segunda forma de la sentencia for

for (($expr_1$; $expr_2$; $expr_3$)) ; do command; done

El ciclo for comienza evaluando la expresión aritmética $expr_1$ (solo cuando se inicia el ciclo, es decir, una vez). Después, se repite el ciclo hasta que la evaluación de la expresión aritmética $expr_2$ sea cero. Para cada evaluación de $expr_2$ que sea distinta de cero, se evalúa la expresión aritmética $expr_3$ y se ejecuta command.

Se puede omitir cualquier expresión aritmética ($expr_1$, $expr_2$, $expr_2$) y, en ese caso, for se comportará como si la evaluación fuera 1.

El valor retornado es el valor de salida del último comando que se ejecute, o es falso si cualquiera de las expresiones no es válida.

El programa V.8 es una variante del programa V.7, donde se usa la segunda forma del for. En este caso, el ciclo se basa en la comprobación del valor numérico de la variable count (debe contener la cantidad de archivos que se van a listar). Los nombres de los archivos quedan almacenados en la variable Files, al igual que en el programa V.7.

```
#!/bin/bash
# Programa V.8

if [ $# -eq 0 ]                  # Si no se pasan parámetros:
  then
    Files=$(ls)                  #-tomar el listado del directorio actual,
    set $Files                   #-fijar las variables posicionales con los nombres de los archivos
  else                           # Si se pasan parámetros:
    Files=$*                     #-asumir que son nombres de archivos
fi

count=$(echo $Files | wc -w)     # Contar los nombres

for((i = 1; i<= $count; i++))    # Para todos los nombres
  do
    if [ -d $1 ]
      then
        echo "$1 es un directorio no se toma en cuenta"
      else
        if [ -e $1 ]
          echo "Contenido del archivo $1:"
          cat $1                 # Si es un archivo, mostrar su contenido
        fi
    fi
    shift                        # Correr a la izquierda las variables posicionales
  done
```

Para contar la cantidad de archivos, se usa la orden count=$(echo $Files | wc -w). En esta acción, se entuba el contenido de la variable Files hacia el comando wc para que cuente las palabras (son los nombres de los archivos, opción -w).

Debe observarse que la sección then del programa V.8 usa la sentencia set $Files para fijar los valores de las variables posicionales con los contenidos de la variable Files. La parte else no necesita hacer esa acción porque se llega a ella si se pasan parámetros, y en ese caso, las variables posicionales quedan fijadas con esos parámetros.

El ciclo for se repite desde 1 hasta la cantidad de nombres (se supone que son nombres de archivos, lo cual no tiene que ser verdadero en el caso de que se pasen parámetros). En la primera iteración, la variable posicional 1 (referida como $1) contiene el nombre del primer archivo en la lista. Al final del ciclo (antes del done), se usa la sentencia shift, de modo que las variables posicionales rotan hacia la izquierda (figura V.4) para así tener siempre el nombre del próximo archivo en la variable posicional 1. La sentencia shift admite un número que especifica la cantidad de posiciones a rotar (por defecto se rota una). Obsérvese que al rotar a la izquierda, los valores de ese extremo se van perdiendo.

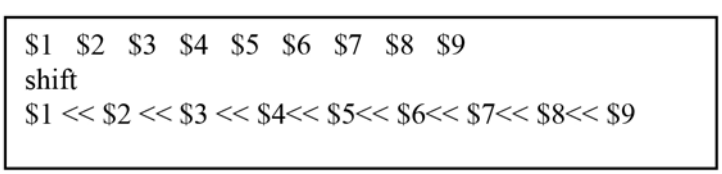


Figura V.4. La sentencia shift rota los argumentos a la izquierda.

Sentencia while

Sintaxis:

while $list_1$; do command; done

La sentencia while ejecuta $list_1$, y si su estado de salida es cero, ejecuta command. Este ciclo de prueba y ejecución se hace hasta que $list_1$ devuelva código de salida distinto de cero.

```
#!/bin/bash
# Programa V.9
echo "Login   Name   Home   Shell" > Users
cat /etc/passwd |
  while  read Data
    do
      login=$(echo  "$Data" | cut -d  : -f 1)
      name=$(echo "$Data" | cut -d  : -f 5 | cut -d , -f 1)
      home=$(echo "$Data" | cut -d  : -f 6)
      shell=$(echo  "$Data" | cut -d  : -f 7)
      echo "$login   $name   $home   shell" >> Users
  done
```

El programa V.9 muestra un ejemplo del uso de la sentencia while. El programa hace un reporte de todos los usuarios que tienen cuenta en una máquina dada. El reporte incluye los siguientes datos para cada usuario: el nombre con que se identifica dentro del sistema (el *login*), su nombre completo, el directorio donde lo sitúa el SO cuando inicia sesión (el directorio de inicio) y el intérprete de comando que usa (el Shell).

El programa V.9 toma los datos del archivo passwd que reside en el directorio /etc. Ese archivo contiene información de todos los usuarios e incluye los siguientes datos: nombre de usuario, nombre real, directorio de inicio (*home*), una referencia a la palabra clave (*password*), el Shell que se usa, etc. El formato del archivo passwd es el siguiente (son siete campos separados por dos puntos): login:clave:uid:gid:info:homeDir:Shell:

El primer campo (login) contiene el nombre con el que se identifica el usuario en el sistema; el segundo campo (clave) contiene una letra x, que indica que la palabra clave está almacenada en el archivo /etc/shadow en forma encriptada; el tercer campo (UID) contiene la identificación del usuario (*user identification*)[67]; el cuarto campo (gid) contiene la identificación de grupo primario (se almacena en /etc/group); el quinto campo (info) contiene información adicional de cada usuario (nombre completo, teléfono, etc.); el sexto campo (homeDir) contiene el camino absoluto hacia el directorio de inicio del usuario (si el directorio no existe, entonces el directorio raíz se convierte en el directorio de inicio); el séptimo campo (Shell) contiene el camino absoluto hacia el intérprete de comandos.

67 El UID cero (0) es del usuario root, los UID del 1 al 99 se reservan para otras cuentas predefinidas y los UID del 100 al 999 se reservan para cuentas de administración y de sistema.

Ahora se puede analizar el programa V.9. En la primera línea, el comando echo redirige su salida (>) hacia un archivo denominado Users, con el objetivo de ponerle un encabezado al listado que se guardará en ese archivo: echo "Login Name Home Shell" > Users

En la instrucción siguiente, la salida del comando cat passwd se entuba (|) hacia la sentencia while (que en realidad es un comando); después, cada iteración de while hace las acciones siguientes: lee una línea del archivo passwd (read Data), se extraen los campos 1, 5, 6 y 7, y se asignan a las variables login, name, home y Shell, respectivamente, para después redirigir esos valores hacia el final del archivo Users (>>). Obsérvese que la orden name=$(echo "$Data" | cut -d : -f 5 | cut -d , -f 1) es diferente a las demás, debido a que el campo 5 de cada entrada del archivo passwd contiene una información variada, separada por comas y solo se desea su primer campo (cut -d , -f 1), es decir, el nombre del usuario (no es lo mismo que el login).

Sentencia until

Sintaxis:

until $list_1$; do $list_2$; command; done

La sentencia until es idéntica a la sentencia while, pero en este caso se ejecuta command hasta que el código de salida de $list_1$ devuelva un valor distinto de cero.

```
#!/bin/bash
#Programa V.10
cmd=$1                              #Asignar el nombre del primer comando a cmd
until [ -z $1 ]
  do
    shift
    until [ -z $1 ]
      do
        if [ `echo "$1" | grep '-'` ]   #Si queda alguna opción (comienzan por -)
          then
            cmd="$cmd $1"           #Se le adiciona la opción a la variable cmd
            shift                   #Rotar los argumentos
          else                      #No quedan más opciones,
            cmd1="$1"               #Asignar el nombre del segundo comando a cmd1
            shift                   #Rotar para que no queden más argumentos
        fi
      done
    $cmd                            #Ejecutar el primer comando con sus argumentos
    done
$cmd1                               #Ejecutar el segundo comando
```

El programa V.10 muestra un ejemplo del uso de la sentencia until. El programa supone que se le pasan los nombres de dos comandos: el primero con todas las opciones que se deseen y el segundo sin opciones. El primer until finaliza cuando no existan más argumentos (se detiene si [-z $1] es verdadero), mientras que el until más interior recorre toda la línea de comando y arma los dos comandos, de modo que la variable cmd

quedará con una cadena que contiene el primer comando, con todas sus opciones, y la variable cmd1 contendrá el segundo comando; ambos se ejecutan con la orden $cmd y $cmd1 respectivamente; por ejemplo, si los argumentos que se le pasan al programa son: ls -l -a ps, la variable cmd contendrá la cadena ls -l -a y la variable cmd1 contendrá la cadena ps.

El estado de salida de las sentencias while y until es el estado de salida del último comando que se ejecute o es cero si no se ejecuta ningún comando.

Ejercicio

Transforme los programas V.7 y V.8 para usar las sentencias while y until en sustitución de la sentencia for.

Sentencia select

Sintaxis:

select name [in $word_1 \ldots word_n$] ; do command ; done

La sentencia select expande la lista de palabras ($word_1 \ldots word_n$) que siguen a in, generando una lista de ítems que se imprimen, en líneas separadas, por la salida de error estándar; las líneas se numeran a partir de 1. Si se omite in word, se imprimen los parámetros posicionales. Debajo de la lista se imprime el símbolo del *prompt* PS3. Después de esas acciones, select queda en espera de que el usuario teclee su selección (seguida de cambio de línea), y después de que se haga una selección, se ejecuta command.

Si la línea leída:

* Contiene uno de los números, la variable name recibe la palabra asociada a esa etiqueta.
* Es vacía, muestra la lista de palabras y el *prompt* de nuevo.
* Si se teclea fin de archivo (ctrl d), el comando termina.
* Si se teclea cualquier otro valor, la variable name toma valor NULL.

La línea leída se guarda en la variable REPLY.

La sentencia select no termina hasta que se ejecute un comando break o se teclee el fin de archivo.

El estado de salida de la sentencia select es el estado de salida del último comando de la lista que se ejecute o es cero si no se ejecuta ningún comando.

El programa V.11 muestra un ejemplo del uso de la sentencia select. El programa no tiene utilidad alguna y solo sirve como un ejemplo, que se hace intencionadamente sencillo.

```
#!/bin/bash
# Programa V.11
# Se fija la variable PS3
PS3="¿A cuál país le gustaría viajar?: "
select Var in Cuba Colombia "Reino Unido"
  do
    echo "Su país preferido para viajar, $Var, es una buena selección."
  done
```

Lo primero que hace el programa es fijar un valor para la variable PS3 y después se usa la sentencia select, a la cual se asocia una lista de tres valores (separados por espacios: Cuba, Colombia, "Reino Unido"). La ejecución de select provoca que se impriman, por la terminal, cada uno de los elementos de la lista, precedidos por un número y un paréntesis cerrado, tras lo cual se imprime el contenido de la variable PS3. Observe la salida del programa V.10 a continuación:

1. Cuba
2. Colombia
3. Reino Unido

   ¿A cuál país le gustaría viajar?

Luego de presentar esta salida, el programa se queda en espera de que el usuario teclee su respuesta y después de leerla ejecuta el comando echo, el cual usa el valor que se asoció a la variable Var para imprimir un mensaje. Si la respuesta es:

* 1, 2 o 3, el programa emite el siguiente mensaje:

  Su país preferido para viajar, <país seleccionado>, es una buena selección.

  En este caso, la variable name se ha asociado a uno de los tres valores posibles: Cuba, Colombia, "Reino Unido".

* El usuario teclea fin de archivo (ctrl d), select finaliza y, como no hay ninguna otra acción en el código, el programa también finaliza.

* Ante cualquier otra respuesta, el programa emite el siguiente mensaje:

  Su país preferido para viajar,, es una buena selección.

En este caso, la variable name quedó con valor NULL, de ahí que el mensaje no contenga el nombre del país.

```
#!/bin/bash
#Programa V.12
PS3="Entre el número asociado a lo que desea saber: "

select Var in "Datos de conexión" "Usuarios conectados"
  do
    case $Var in
      'Datos de conexión')
        Data="$(ifconfig | grep 'inet addr:' | grep -v 127)"
        IP=`echo $Data | cut -d : -f2 | sed s/Bcast//`
        Bcast=`echo $Data | cut -d : -f3 | sed s/Mask//`
        Mask=`echo $Data | cut -d : -f4`
        echo "Dirección IP: $IP, bcast $Bcast, máscara $Mask";;
      'Usuarios conectados')
        #Se deja de ejercicio;;
      *)
        echo "Entrada no válida"
        exit 1;;
    esac
  done
# Fin del programa
```

El programa V.12 es un poco "más complejo" y hace algo útil. Su propósito es mostrar alguna información acerca del sistema, para lo cual ofrece varias opciones que se listan por medio de la sentencia select.

El usuario debe elegir una de esas opciones, y luego la variable Var se instancia con el valor de la palabra asociada a la opción elegida; por ejemplo, si el usuario teclea 1, la variable Var tomará el valor 'Datos de conexión'. Después de que el usuario elige su respuesta, se ejecuta la sentencia case, la cual contiene una etiqueta por cada una de las palabras que le siguen a la palabra in en la sentencia select, de modo que si el usuario tecleó 1, la sentencia case ejecuta todas las instrucciones que están asociadas a la etiqueta 'Datos de conexión'.

A continuación, se analizan todas las acciones asociadas a la etiqueta 'Datos de conexión', de la sentencia case del programa V.12.

En el comando Data="$(ifconfig | grep 'inet addr:' | grep -v 127)", se le asigna a la variable de usuario Data el resultado de la ejecución del comando ifconfig. Este comando brinda varias líneas de información, pero solo interesa en este caso conocer datos relacionados con el direccionamiento ip, por eso su salida se entuba dos veces (con el propósito de filtrarla):

* El primer entubamiento es hacia el comando grep (grep 'inet addr:'), el cual hace un filtraje para quedarse solo con las líneas que contengan la cadena 'inet addr:'; son dos, por ejemplo (no tienen que coincidir con las que obtenga el lector):

```
inet addr:10.0.2.15 Bcast:10.0.2.255 Mask:255.255.255.0
inet addr:127.0.0.1 Bcast:255.0.0.0
```

* Después, el nuevo resultado se entuba hacia grep -v 127 para desechar las líneas (opción -v) que contengan 127. De modo que la variable Data solo recibe la cadena asociada a la dirección IP básica, es decir, la primera.

El segundo comando de esta etiqueta hace un filtraje para dejar en la variable IP la dirección ip de la máquina. Para esto, se usa el comando echo que imprime el contenido de la variable Data y lo entuba hacia el comando cut, el cual toma como delimitador al signo de dos puntos (-d :) para elegir el segundo campo (-f 2) de la cadena (10.0.2.15 Bcast). Ese resultado se entuba de nuevo hacia el comando sed (sed s/Bcast//), el cual sustituye la cadena Bcast por nada (s/Bcast//), obteniendo la dirección ip (10.0.2.15).

En los dos comandos siguientes, se obtienen (de forma similar) la dirección de mensajes de difusión (*broadcast*) y la máscara. Por último, se imprime la información contenida en las variables (IP, Bcast y Mask).

V.3.3.1 Sentencias que afectan el comportamiento de los ciclos

Las sentencias break y continue afectan el flujo de ejecución de los ciclos, debido a que producen un salto desde la posición actual hacia otra.

* Sintaxis de break:

```
break [n]
```

El programa V.13 no tiene mucha utilidad, el único objetivo es percatarse del efecto que produce la sentencia break. Obsérvese que aunque el ciclo más exterior se debe repetir hasta el valor 12, el uso de break en el ciclo interior provoca que solo se impriman los valores del 1 al 4.

```
#!/bin/bash
#Programa V.13
ctrl=0
while [[ $ctrl -le 12 ]]
  do
    echo $ctrl
    if [ ctrl -eq 4]
      then
        break
    fi
    ((ctrl++))
  done
```

* Sintaxis de continue:

```
continue [n]
```

```
#!/bin/bash
#Programa V.14
ctrl=0

while [[ $ctrl -le 12 ]]
  do
    echo $ctrl
    if [ ctrl -eq 4]
      then
        ((ctrl++))
        continue
    fi
    ((ctrl++))
  done
```

La sentencia continue salta al inicio del ciclo o salta al inicio del ciclo n veces para hacer una nueva iteración, abandonando la iteración actual.

El programa V.14 es una modificación del programa V.13, pero se ha sustituido la sentencia break por continue y es necesario poner la orden ((ctrl++)) dentro de la sección then de la sentencia if o se producirá un ciclo infinito que constantemente imprimirá 4.

V.3.2 Arreglos

Un arreglo en Bash es una variable que contiene múltiples valores, y a diferencia de otros lenguajes, pueden contener valores de diferentes "tipos" y sus elementos no tienen que estar contiguos.

El índice inferior de los arreglos es cero y no existe un límite para el índice superior. Los arreglos en Shell script solo pueden tener una dimensión.

* Sintaxis de declaración:

```
name[expr]=value
name=(lista de valores separados por espacios)
declare -a name
```

Donde name es el nombre del arreglo y expr es una expresión que tiene que dar como resultado un valor entero.

Ejemplo:

```
Array[2]=12; Array[4]=10
```

En la primera orden, se declara el arreglo Array y se le asigna a su tercer elemento (índice 2) el valor 12. A continuación, se le asigna al quinto elemento (índice 4) el valor

10. Debe observarse que en el arreglo Array se han dejado elementos sin inicializar y por tal motivo esos elementos tienen el valor NULL.

Para referirse al contenido del arreglo, se usa la notación de llaves; por ejemplo, el siguiente fragmento de código imprime el arreglo Array, inicializado antes:

```
for((i=0; i <=5; i++))
  do
    echo -n "Array[$i] es ${Array[i]}, "
  done
```

La salida del ejemplo anterior es la siguiente:

```
Array[0] es , Array[1] es , Array[2] es 12, Array[3] es ,Array[4] es 10
```

Obsérvese que los elementos Array[0], Array[1] y Array[3] no se han inicializado y por eso contienen el valor NULL, de ahí que aparezcan los espacios después de la palabra es en los lugares que corresponderían a los valores de esos elementos del arreglo.

Se pueden asignar valores iniciales a múltiples elementos de un arreglo usando una lista de elementos entre paréntesis separados por espacios, por ejemplo:

```
B=(1 otro nuevo 123)
```

A diferencia de muchos otros lenguajes, los arreglos en Bash pueden contener distintos "tipos de datos". Observe el programa V.15 en el que se usan dos ciclos for: el primero imprime los elementos del arreglo B y el segundo multiplica cada elemento por tres, B[$i]*3.

```
#!/bin/bash
#Programa V.15
B=(1 otro nuevo 123)                    #Se usa una lista para asignar los valores al arreglo B
echo "Valores iniciales del arreglo B"  #Imprimir un título para esta parte de la salida
for((i=1; i<=3; i++))                   #Recorrer el arreglo B imprimiendo sus valores
  do  echo "B[$i]=${B[i]}" done
echo "Nuevos valores del arreglo B"
for((i=1; i<=3; i++))
  do
    B[i]=$((B[i]*3))                    #Realizar una operación aritmética sobre cada elemento
    echo "B[$i]=${B[i]}"
  done
```

La salida del programa anterior será la siguiente:

```
Valores iniciales del arreglo B
B[0]=1 B[1]=otro B[2]=nuevo B[3]=123
Nuevos valores del arreglo B
B[0]=3 B[1]=0 B[2]=0 B[3]=369
```

Debe observarse que los nuevos valores de los elementos B[1] y B[2] son cero, debido a que no es posible multiplicar una cadena por un número.

Para determinar la cantidad de elementos de un arreglo, se usan las expresiones ${#name[@]} y ${#name[*]}, donde name es el nombre del arreglo. Debe señalarse que esa longitud está determinada por la cantidad de elementos que se han inicializado y no por el índice mayor. Observe el siguiente segmento de código:

```
B[0]=12; B[10]=23
echo "La longitud del arreglo B calculada usando la forma \${#b[@]} es ${#b[@]}"
echo "La longitud del arreglo B calculada usando la forma \${#b[*]} es ${#b[*]}"
```

La salida de este segmento de código será la siguiente:

```
La longitud del arreglo B calculada usando la forma ${#b[@]} es 2
La longitud del arreglo B calculada usando la forma ${#b[*>]} es 2
```

Es decir que el arreglo B tiene solo dos elementos, pero no están contiguos. Este no es el comportamiento usual de los arreglos en otros lenguajes y por eso no se podría hacer un ciclo for en la forma for((i=0; i<=2; i++)) para referirse a sus elementos, tomando en cuenta el valor de la variable de control de ciclo i.

```
#!/bin/bash
#Programa V.16. Índices y dimensión de los arreglos
M=(La programación es bella)              #  Se asignan cuatro valores al arreglo M,
N=(${M[@]})                               #  Se asigna el arreglo M al arreglo N,
                                          #  es necesario usar paréntesis en los dos casos
F=${M[@]}                                 #  Se asignan los elementos de M a F como una
                                          #  sola cadena, no se pueden usar paréntesis
echo "Los ${#M[@]} elementos del arreglo M son: ${M[@]}"
echo "Los ${#N[*]} elementos del arreglo N son: ${N[*]}"
echo "El primer elemento de M es: ${M[0]}, que contiene ${#M[0]} caracteres"
echo "El segundo elemento de N es: ${N[1]}, que contiene ${#N[1]} caracteres"
echo "El arreglo F solo tiene ${#F[@]} elemento con ${#F[0]} que es \"${F[0]}\""
if [ ${#M[@]} -eq ${#N[@]} ]              #Si la cantidad de elementos de M es igual a la de N
 then
  echo "Los arreglos M y N tienen la misma cantidad de elementos"
fi
exit 0
```

En el programa V.16, se han usado algunas facilidades que se comentan en el mismo lugar donde se usan. Deben hacerse las siguientes observaciones adicionales:

1. Siendo name el nombre de un arreglo y n un número entero:
   a. La forma sintáctica ${name[n]} permite acceder al contenido del elemento n del arreglo name, por ejemplo: ${M[2]}, ${N[3]}, ${F[0]}.
   b. La forma sintáctica ${#name[n]} se usa para conocer la cantidad de caracteres de un elemento de arreglo, por ejemplo: ${#M[2]}, ${#N[3]}, ${#F[0]}.

   c. Las formas sintácticas ${name[@]} y ${name[*]} permiten referirse a todos los elementos de un arreglo, por ejemplo: ${M[@]}, ${N[*]}, ${F[@]}.

   d. Las formas sintácticas ${#name[@]} y ${#name[*]} permiten conocer la cantidad de elementos de un arreglo, por ejemplo: ${#M[@]}, ${#N[*]}, ${#F[@]}.

2. La sentencia M=(La programación es bella) asigna a la variable M una lista de cuatro componentes, separados por espacios, que conforman los elementos del arreglo M. Esa asignación transforma a M en una variable de arreglo.

3. La sentencia N=(${M[@]}) asigna a la variable N la lista de elementos (se usan paréntesis) del arreglo M, por eso ambos arreglos tienen los mismos elementos.

4. La sentencia F=${M[@]} asigna a la variable F los elementos del arreglo M como un todo (no se usan paréntesis), por eso el arreglo F solo contiene un elemento.

Esta sección se termina con el programa V.17, el cual usa otras facilidades para el trabajo con arreglos. El programa tiene una buena cantidad de comentarios[68]; no obstante, a continuación se detallan algunas cosas:

1. Cada línea de salida del comando who produce algo como lo siguiente:
   usuario terminal año-mes-día.

2. Esa salida se entuba hacia el comando cut con la orden: who | cut -d'-' -f 1. Esto hace que cut filtre la entrada escogiendo su primer campo (*field*), -f 1; los campos estarán separados por el delimitador (*delimiter*) guion (-), -d'-'. Lo anterior hace que en la orden: login=(`who | cut -d'-' -f 1`) se construya un arreglo (observe los paréntesis) que tendrá los elementos en la forma: usuario terminal año, es decir, los nombres de los usuarios, la terminal desde la cual están conectados al sistema y el año (primer campo de la fecha en que se conectaron).

3. Después de haber obtenido este primer arreglo, se ejecuta el comando: login=(`echo "${login[*]/2016/}"`), el cual reconstruye el arreglo login eliminando todos los campos que contienen el año (en este caso, 2016), de modo que los elementos del arreglo toman ahora la forma: usuario terminal.

4. El ciclo for, for((i=0; i<size; i=$i+2)), recorre el nuevo arreglo login, de dos en dos, para imprimir en cada iteración: el identificador del usuario ${login[$i]} y la terminal desde la que está conectado ${login[$i+1]}.

68 Aunque es importante comentar los programas, es imprescindible programar lo más claro posible, de manera que el código se pueda leer fácilmente.

```
#!/bin/bash
#Programa V.17. Filtrando elementos de arreglos

login=(`who | cut -d '-' -f 1`)                  #Crear un arreglo con datos de usuarios conectados
echo "login, de longitud ${#login[@]} es:"       #Imprime la longitud del arreglo login
echo "${login[*]}"                               #Imprime el arreglo login
login=(`echo "${login[*]/2016/}"`)               #Se reconstruye login eliminando el campo 2016
size=${#login[@]}                                #size recibe la longitud del nuevo arreglo login
echo "El nuevo login, de longitud $size es:"     #Imprime la longitud del nuevo arreglo login
echo "${login[*]}"                               #Imprime el nuevo arreglo login

#Recorrer los elementos de login de dos en dos para imprimir:
#login y terminal de los usuarios conectados
for((i=0; i<size; i=$i+2))
  do
    echo "Usuario: ${login[$i]}, conectado desde: ${login[$i+1]}"
  done
```

V.3.3 Funciones

El lenguaje Bash también permite definir funciones de usuarios. La definición de una función debe preceder a su llamada para que el intérprete conozca, antes de la llamada, el lugar al que deberá saltar.

* Sintaxis de definición:

  function name { ... }

  name () { ... }

* Donde name es el nombre de la función.

Los parámetros pasados a una función se refieren dentro de ella en forma de parámetros posicionales y solo se pasan por valores (aunque hay algunos "trucos" para pasarlos por referencia). El programa V.18 muestra un ejemplo elemental para trabajar con funciones que reciben parámetros.

```
#!/bin/bash
#Programa V.18. Trabajo con funciones
f()                                  #Función f. Es necesario definirla antes de llamarla
  {                                  #debido a que el Shell es un intérprete
    if [ $# -eq 0 ]                  #Si no se pasan parámetros
      then
        echo "La función no recibió parámetros"
      else                           #Si se pasan parámetros
        echo "La función recibió $# parámetros que son los siguientes:"
        until [ -z $1 ]              #Hasta que no hayan parámetros
          do
            echo -n "$1 "            #Imprimir el parámetro 1 actual
            shift                    #Rotar los parámetros a la izquierda
          done
        echo
    fi
  }

#Seguidamente se  hacen varias llamadas a la función f()
f                                    #Llamada sin parámetros
f $1 $2 $3                           #Llamada suponiendo que el programa recibió 3 parámetros
set `date`                           #Fijar los parámetros posicionales con la salida de date
f $1 $2 $3 $4 $5 $6                  #Llamada con los nuevos valores fijados por date
```

Las funciones siempre retornan un valor que se denomina "estado de salida". El valor implícito que se retorna es el estado de salida del último comando que se ejecute, pero se puede retornar un valor explícitamente si se usa la sentencia return.

El valor del estado de salida de una función se puede usar dentro de un programa script a través de la variable especial ?. Debe recordarse que esa variable también devuelve el estado de salida del programa script.

V.4 RESUMEN DEL CAPÍTULO

Los lenguajes Shell script que acompañan a los SO tipo Unix se caracterizan por la diversidad de herramientas que ofrecen.

Otros SO proporcionan herramientas similares a los lenguajes Shell script de los SO tipo Unix, pero la potencia de estas herramientas en Unix se considera de las más poderosas.

Existen varios Shell que se pueden usar en los SO tipo Unix, por ejemplo: Bash shell, Bourne shell, C-shell, Korn shell, etc. Asociado a cada uno de ellos, existen un lenguaje Shell script que posee características similares a los demás Shell, pero en esencia son distintos. De ahí que es muy recomendable iniciar todo programa Shell script con la línea especial #!, que especifica el Shell que deberá interpretar el programa.

El Bash es el lenguaje Shell script que se ha usado en este capítulo debido a que, es quizás el más popular y versátil de esos lenguajes entre los SO tipo Unix (es una opinión personal del autor con la cual se puede estar en desacuerdo).

Cuando se hace un programa en cualquier lenguaje de programación, debe ponerse sumo cuidado en su claridad, pues si no se hace así, las futuras modificaciones o ampliaciones se pueden tornar complejas y puede incluso que la propia persona que lo hizo no se percate del porqué de muchas decisiones tomadas. Es recomendable:

* Identar todo el programa de una manera homogénea y clara, como cuando se escribe un texto cualquiera.
* Usar convenios estándares para los nombres de las variables, de modo que para el caso del Shell script se pueda distinguir entre las variables: de usuario, de ambiente y posicionales.
* Poner comentarios adecuados que faciliten la lectura del programa en un futuro.

En el lenguaje Shell script, se pueden usar diversas facilidades; cabe destacar las siguientes:

* Manipular procesos, redireccionar entrada/salida, usar diversos filtros, etc.
* Programar en un lenguaje sencillo y propio del SO, que se auxilia de facilidades muy similares a la de algunos lenguajes de alto nivel. Entre esas facilidades se pueden citar las siguientes:
  - Uso de operadores de distintos tipos.
  - Uso de sentencias diversas (condicionales, de repetición, de asignación, etc.).
  - Trabajo con arreglos.
  - Trabajo con funciones.

En este capítulo, se ha hecho un estudio sintetizado del lenguaje Shell script. Existen diversos libros que estudian el lenguaje más ampliamente, aunque algunos de ellos se enfocan en la descripción del lenguaje en sí y los ejemplos que se citan no son muy prácticos.

Aunque el capítulo no es muy amplio, se ha tratado de que los ejemplos sean aplicables en entornos de SO, debido a que el libro es para enseñar precisamente esa materia. No se finaliza este capítulo con una sección de ejercicios propios, como se hizo en los restantes, dado que el planteamiento de ejercicios de programación en entornos de SO puede hacerse complejo, y se ha preferido emitir las siguientes recomendaciones:

1. Busque los programas Shell script que acompañan a su SO tipo Unix. Estúdielos profundamente, de modo que esté consciente de que los ha entendido.

   Nota. En su directorio de inicio encontrará algunos. Si está usando el Bash, podrá ver los archivos ocultos .bash_history, .bash_logout, bashrc y en el directorio /etc encontrará algunos otros, tales como bash.bashrc, que son complementos de los anteriores.

a. Modifique esos programas y adáptelos a su conveniencia.

b. Verifique su funcionamiento.

2. Trate de automatizar, a través de programas hechos en Bash, algunas de las tareas rutinarias que se hacen en un entorno Unix. Estas pueden ser:

a. Hacer cuentas de usuario para un grupo de personas. Esa es una tarea habitual que se realiza en cualquier centro de estudios cuando comienza un curso.

b. Fijarle cuotas de uso de espacio en disco a los usuarios, etc.

V.5 BIBLIOGRAFÍA CONSULTADA

Cooper, M. *Advanced Bash-Scripting Guide. An in-depth exploration of the art of shell scripting*. Recuperado el 24 de febrero de 2016, de http://www.tldp.org/LDP/abs/abs-guide.pdf

Garrels, M. *Bash Guide for Beginners*. Recuperado el 24 de febrero de 2016, de http://tille.garrels.be/training/bash/

Gilly, D. (2003). *Unix in a Nutshell*. En *The Unix CD Bookshelf* (3.ª ed.). Sebastopol: O'Reilly Media.

Kochan, S. G., & Wood, P. H. (2003). *UNIX Shell Programming* (3.ª ed.). Indianápolis: Sams Publishing.

Love, P., Merlino, J., Red, J. C., Zimmerman, C., & Weinstein, P. (2005). *Beginning UNIX*. Nueva York: John Wiley & Sons.

Mohr, J. (2001). *Linux-user's resource*. Nueva Jersey: Prentice Hall.

Peek, J., O'Reilly, T., & Loukides, M. (2003). *Unix Power Tools*. En *The Unix CD Bookshelf* (3.ª ed.). Sebastopol: O'Reilly Media.

Peek, J., Todino G., & Strang, J. (2003). *Learning the Unix Operating System*. En *The Unix CD Bookshelf* (3.ª ed.). Sebastopol: O'Reilly Media.

Petersen, R. (2009). *Manual de referencia Linux* (6.ª ed.). Nueva York: McGraw-Hill.

Robbins, A., & Lamb, L. (2003). *Learning the vi Editor*. En *The Unix CD Bookshelf* (3.ª ed.). Sebastopol: O'Reilly Media.

Robbins, K., & Robbins, S. (2003). Unix Systems Programming: Communication, concurrency, and threads. Nueva Jersey: Prentice Hall.

Rosenblatt, B. (2003). *Learning the Korn Shell*. En *The Unix CD Bookshelf* (3.ª ed.). Sebastopol: O'Reilly Media.

Shotts Jr., W. (2012). *The Linux Command Line: A complete introduction*. San Francisco: No Starch Press.

Tiemann, B., & Urban, M. C. (2001). *FreeBSD Unleashed*. Indianápolis: Sams Publishing.

Conclusiones

Los SO actúan como una interfaz de comunicación entre los usuarios de la computadora y el *hardware*, presentando un medio cómodo, eficiente y seguro para explotar los recursos de la computadora, sin tener que conocer los detalles específicos de cada uno de los componentes del sistema de cómputo.

Los SO modernos se diseñan por partes o módulos, que tienen responsabilidades específicas e interactúan entre sí para resolver las diversas tareas que tienen a su cargo. En general, se destacan tres subsistemas: de planificación del procesador, de administración de la memoria y de archivos. El SO es el más importante programa de sistema (programa que da apoyo a otros programas).

El subsistema de planificación del procesador trabaja fundamentalmente con procesos. Los procesos son **entes activos** y, a diferencia de los programas que son entes pasivos, se puede decir que tienen vida, debido a que son **programas en ejecución** que poseen recursos (de *software* y de *hardware*) y realizan diversas acciones. Dentro del SO, los procesos están representados por una estructura de datos, denominada **Bloque de Control de Proceso** (PCB), en la cual se mantienen todos los datos del proceso. Los planificadores y el despachador usan el PCB para realizar su trabajo.

Existen diversos algoritmos para planificar el uso del procesador, los cuales pueden ser **con desalojo** y **sin desalojo**. La vida de los procesos transcurre por diferentes estados, que pueden ser: listo, ejecutando, terminado y bloqueado, entre otros.

Si dos o más procesos comparten recursos que se deben usar en forma exclusiva, se deberán tener cuidados especiales para que se puedan ejecutar en forma concurrente. Los segmentos de código donde se deben tener esos cuidados se denominan secciones críticas, y existen mecanismos de *software* y de *hardware* para controlar el uso de los recursos compartidos en esas secciones.

El sistema de archivo es un subsistema dentro del SO que se encarga de administrar los volúmenes de almacenamiento. Para facilitar el trabajo del SO y de los usuarios, los archivos se organizan en directorios.

Existen tres técnicas generales para asignar espacios dentro de un sistema de archivo: contigua, enlazada e indexada. Esta última es la más usada hoy en día y los sistemas NTFS y ExtX se destacan dentro de los SO Windows y Linux, respectivamente.

La memoria es un recurso muy importante y para lograr altos grados de multiprogramación es necesario compartirla. A fin de que la memoria se pueda repartir de

manera eficiente, existen diversas técnicas de administración, entre las que se destacan el paginado y la segmentación.

Cuando se trabaja con memoria virtual, tanto el paginado como la segmentación se auxilian de diferentes algoritmos para escoger, entre las partes cargadas en memoria, las que serán sustituidas por otras. Los sistemas segmentados sufren de fragmentación externa, mientras que los sistemas paginados padecen de fragmentación interna.

Existen diversas implementaciones de los SO al estilo o tipo Unix, pero todas las distribuciones poseen características comunes que permiten estudiarlas en una forma genérica. Estos SO son: multiusuarios, de multiprocesamiento y multitareas.

Los SO tipo Unix vienen acompañados de diferentes intérpretes de comandos o Shell, como pueden ser: el Bourne Shell, el Bourne Again Shell y el C Shell, entre otros. Los usuarios pueden elegir cuál de ellos usarán, dado que el Shell en sí no forma parte del SO.

El Shell no es solo un intérprete de comandos, debido a que viene acompañado de un potente lenguaje de programación, conocido como Shell script. Entre esos lenguajes se destaca el del intérprete del Bash.

En el lenguaje Bash script, se pueden usar varias facilidades que están presentes en cualquier lenguaje de programación de alto nivel y, además, se pueden explotar las diferentes posibilidades que brinda el Shell.

En este libro, se ha hecho un estudio general de los SO. Cualquiera de las partes estudiadas puede también ser parte de un nuevo libro y por eso el lector estudioso debería profundizar en los diversos temas que se han discutido, para lo cual el presente estudio es solo un punto de partida.

Anexos

Anexo A. Sistemas operativos sobre máquinas virtuales

Existen diversos *software* para crear máquinas virtuales. En esta guía, se hace referencia al VirtualBox porque es un *software* libre[69], lo que significa que se puede explorar su código fuente y transformarlo según las necesidades que se requieran; también, que está sujeto a licencias que no son muy restrictivas. Debe observarse que el hecho de que un *software* sea libre no significa que sea gratis, aunque el VirtualBox sí lo era al momento de escribir este libro y posiblemente aun lo sea cuando el lector esté leyendo estas notas.

Lo primero que debe hacerse es instalar el VirtualBox, lo cual no es nada complejo dado que el sistema viene acompañado por un asistente (*wizard*).

La figura A.1 muestra la interfaz de usuario del sistema VirtualBox ya instalado. En ese momento, no se ha creado ninguna máquina virtual.

Figura A.1. El VirtualBox.

Nota: Todas las imágenes que acompañan los anexos se captaron al actuar directamente con los sistemas. Se utilizó la versión 5.014 del VirtualBox y la 8.3.0 del SO Debian.

69 Libertades del *software* libre. Libertad 0: se puede usar con cualquier propósito. Libertad 1: permite estudiar su funcionamiento, modificarlo y adaptarlo a necesidades propias. Libertad 2: se pueden distribuir copias del programa. Libertad 3: se puede mejorar el programa y publicar la versión mejorada.

A.1 GUÍA DE INSTALACIÓN DE UNA MÁQUINA VIRTUAL SOBRE VIRTUALBOX

Los pasos descritos a continuación explican los detalles para instalar una máquina virtual sobre el hipervisor o monitor de máquinas virtuales VirtualBox.

1. Oprima el botón izquierdo del *mouse* sobre el botón **Nueva** para crear la nueva máquina virtual (observe la figura A.1). Después, deberá ver algo similar a la figura A.2. Llene los campos en forma adecuada, observe que: el primer campo le da nombre a la máquina virtual que se creará (puede ser cualquiera), mientras que los campos restantes deben ser llenados de acuerdo con el SO que se haya escogido para instalar posteriormente sobre la máquina virtual.

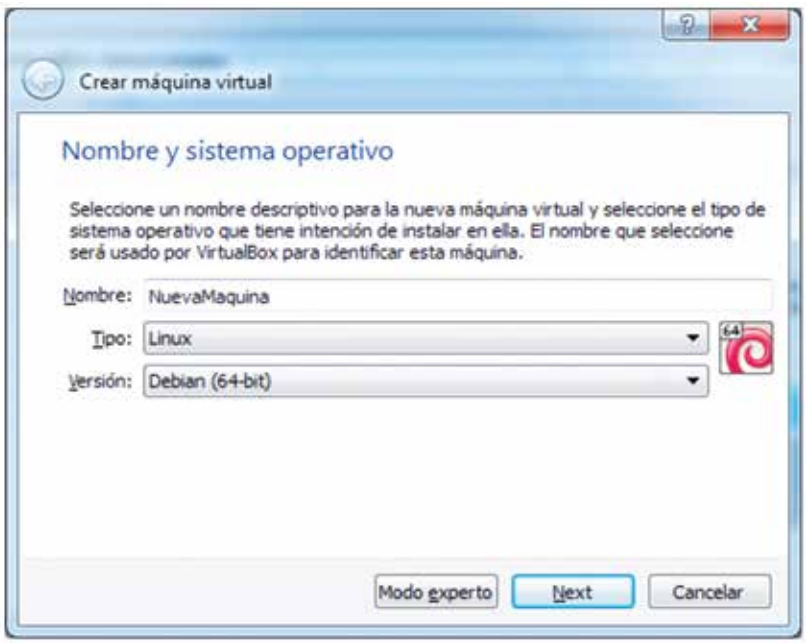


Figura A.2. Creando una máquina virtual sobre VirtualBox.

En este caso, se instalará un SO Debian de 64 bits que pertenece a la familia de SO Linux. Llene los campos en forma adecuada y oprima el botón **Next**.

2. Una vez hecha la acción anterior, se presenta algo similar a la figura A.3, donde se permite especificar la cantidad de memoria que tendrá la máquina virtual. El sistema propone una cierta cantidad que se puede aumentar o disminuir, pero esa memoria se toma de la memoria física (real) que tiene la máquina hospedera, de modo que no puede ser mayor que la cantidad real y ni siquiera muy cercana a esa cantidad, porque se estaría dejando a la máquina hospedera sin memoria. En esta guía, se propone dejar la cantidad propuesta por el VirtualBox, pero la práctica le ayudará a determinar cuál es la cantidad adecuada para sus propósitos.

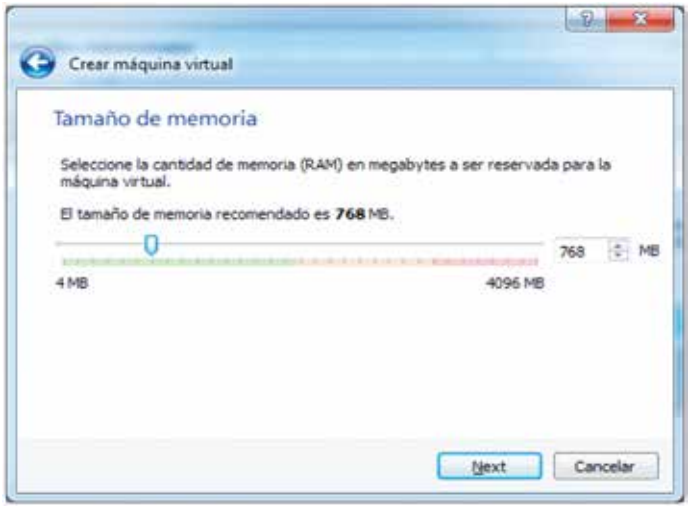


Figura A.3. Especificando la cantidad de memoria.

Escoja la cantidad que estime o deje la propuesta y oprima el botón **Next**.

3. Una vez que se ha especificado el tamaño de la memoria, debe establecerse el disco duro de la máquina virtual, puede ser un disco nuevo o un disco que ya exista. En esta guía, se creará un disco nuevo (figura A.4).

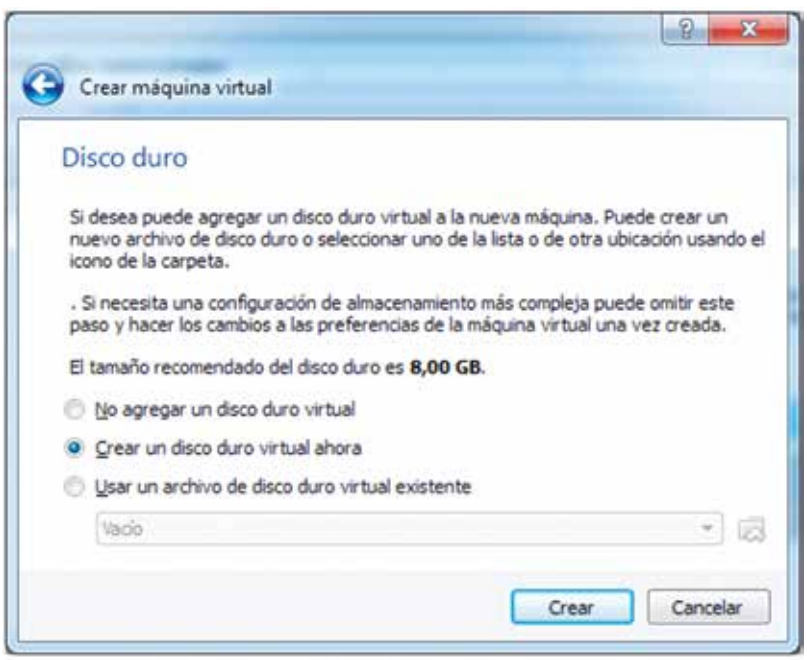


Figura A.4. Creando un disco duro virtual.

Los discos se emulan sobre un archivo, de modo que no es necesario preocuparse por su tamaño. Marque la opción "Crear un disco duro virtual ahora" y oprima el botón **Crear**.

Al finalizar el proceso de creación de la máquina virtual y la instalación del SO, podrá usarse el archivo que emula el disco en cualquier otra máquina hospedera, para lo cual bastará que en este paso se escoja la opción "Usar un archivo de disco duro virtual existente".

4. Ahora se puede escoger el tipo de archivo que se usará para el disco virtual, lo cual dependerá del *software* que se use para manipular la máquina virtual. En esta guía, todo se hace con VirtualBox, por eso la opción escogida es la primera. Observe la figura A.5. Oprima el botón **Next**.

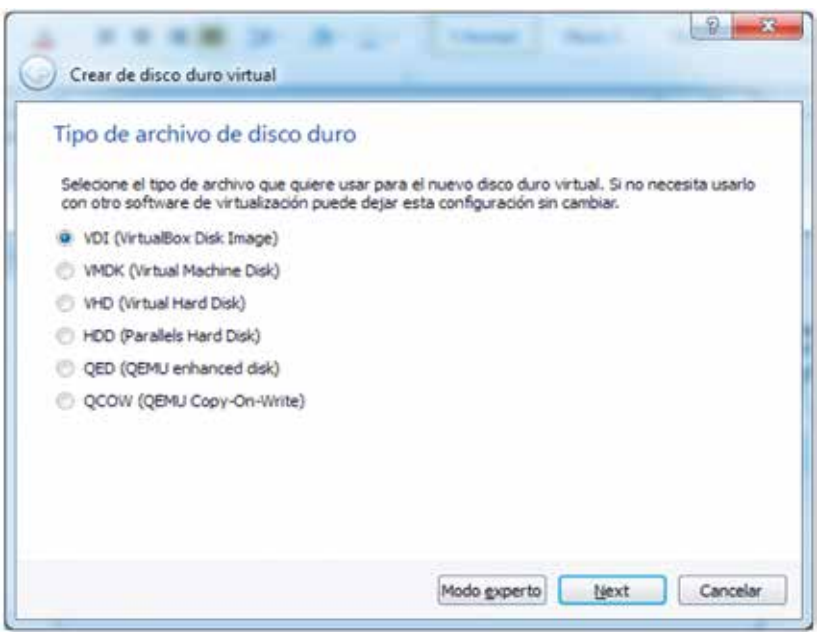


Figura A.5. Selección del tipo de archivo para el disco virtual.

5. Según la ventana que se aprecia en la figura A.6, existen dos formas de crear un disco virtual:

   a. Prefijando el tamaño. En ese caso, el proceso demora un poco y además el archivo tiene ese tamaño aun cuando el disco esté completamente vacío.

   b. De expansión dinámica. Permite definir un disco con un tamaño máximo que se inicia con poco espacio y crece, dinámicamente, de acuerdo con las necesidades y hasta el límite que se haya fijado. El crecimiento dinámico se puede hacer gracias a que el "disco" es en realidad un archivo y no un disco físico. En esta guía, se usa el crecimiento dinámico. Marque esa opción y oprima el botón **Next**.

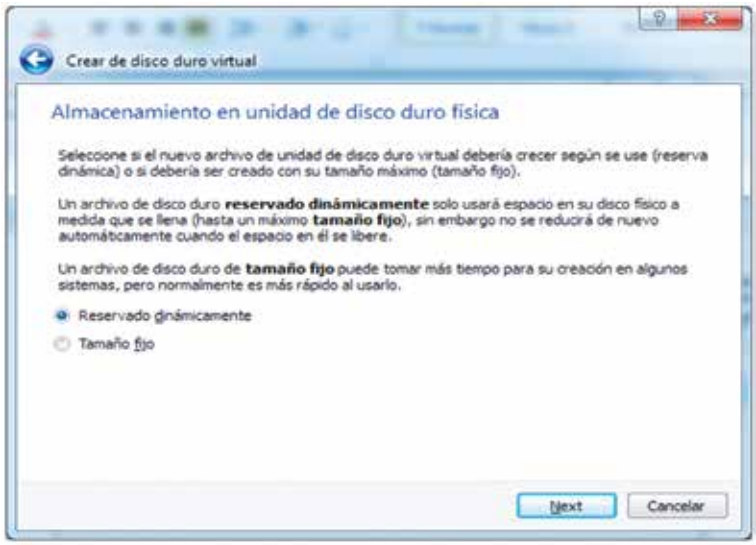


Figura A.6. Creando un disco virtual de crecimiento dinámico.

6. El próximo paso es darle un nombre al disco que se desea crear (figura A.7). También se puede usar el botón de navegación, en forma de flecha, para poner el nombre y escoger la localización que tendrá el archivo que soporta el disco virtual. Se permite, además, utilizar el botón deslizable o el campo numérico a la derecha de ese botón para fijar el tamaño inicial del disco (observe la figura A.7).

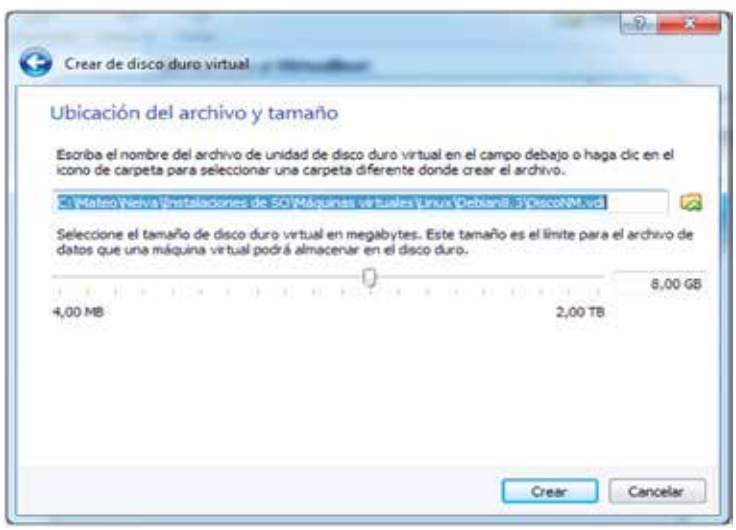


Figura A.7. Estableciendo características del disco virtual.

En este caso, el disco se nombra DiscoNM.vdi, se localiza en el directorio Debian8.3 (observe el camino completo en la figura) y su tamaño máximo será 8 GB.

Oprima el botón **Crear** para finalizar esta parte del proceso.

7. Ya queda la máquina virtual creada (figura A.8) con las características que se le especificaron.

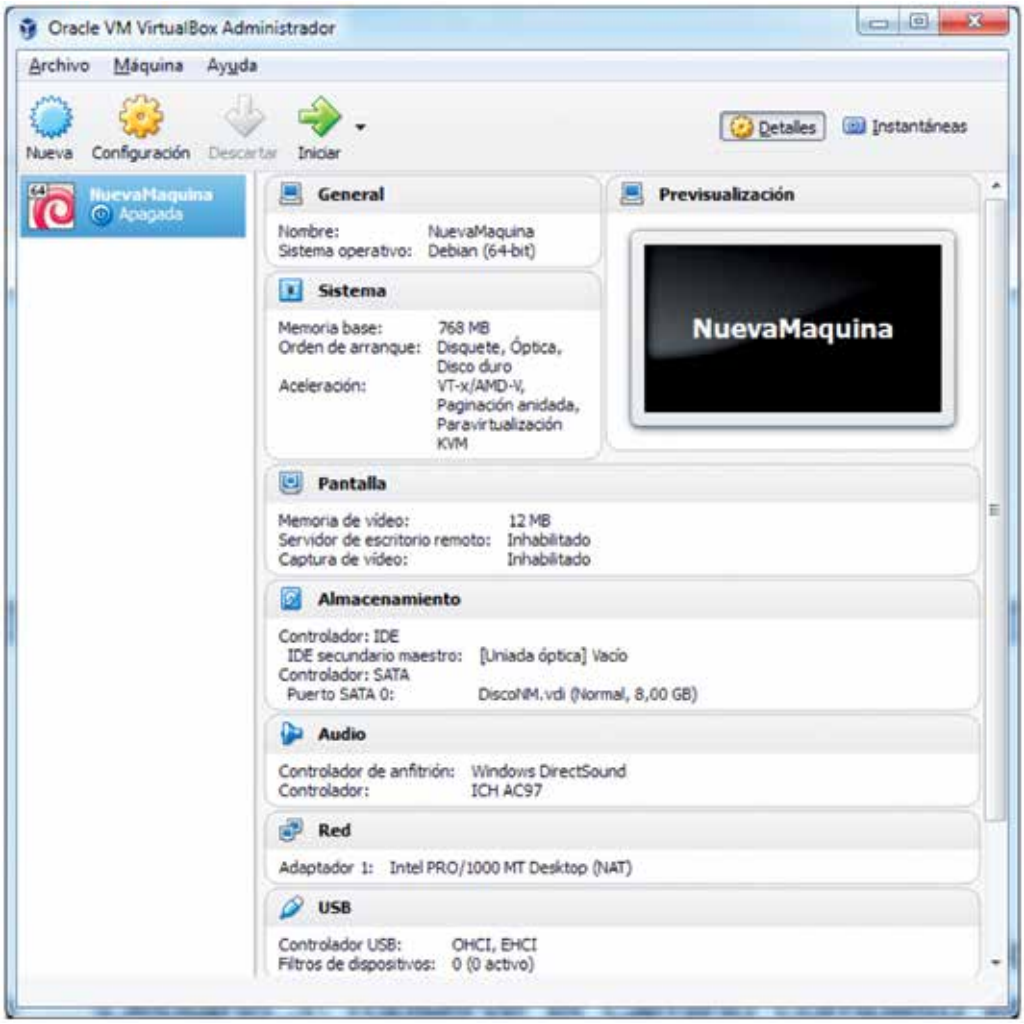


Figura A.8. Máquina virtual NuevaMaquina en estado de apagada.

La máquina creada está en el estado de apagada y si se enciende, no funcionará porque no tiene un SO instalado.

A.2 GUÍA DE CONFIGURACIÓN DE LA MÁQUINA VIRTUAL PARA INSTALAR UN SO

Esta parte es general y, excepto por algunas especificaciones que resultarán obvias, se puede usar para instalar cualquier SO sobre el VirtualBox.

Se usará un CD virtual para instalar el SO sobre la máquina virtual que se creó en la guía anterior. El CD virtual contiene la imagen de instalación del SO en forma de un archivo ISO[70].

70 Un archivo ISO o imagen ISO contiene una representación exacta de un CD o un DVD.

Lo primero que hay que hacer para instalar un SO desde un dispositivo es configurar el CMOS[71] para que el proceso de arranque se haga desde ese dispositivo. Después de esto, si se va a hacer desde un dispositivo externo, hay que insertarlo en la máquina. En esta guía, se usará un CD que no es físico (es virtual) y por eso no se pondrá el CD en la disquetera del equipo, pero se efectuará la acción equivalente con el CD virtual.

1. Con la máquina apagada, oprima el botón **Configuración** y después elija **Sistema** (observe la figura A.9). El propósito es fijar el orden de arranque que intentará la máquina.

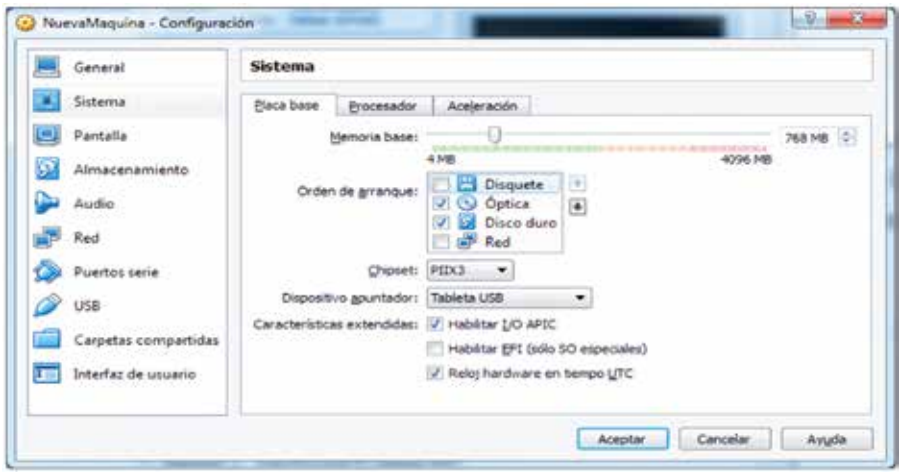


Figura A.9. Configurando el arranque de la máquina virtual.

En la figura A.9, se ha configurado la máquina virtual MiMaquina para que siga el orden de arranque **Óptica**, **Disco duro**. El orden preestablecido tenía **Disquete** como primera opción.

Una vez establecido que el arranque se intenta primero desde el CD, oprima el botón **Aceptar**.

2. Ahora es necesario poner el CD en la disquetera. En este caso, es un CD virtual y por eso no se hará esa acción física; en su lugar, se configura la máquina para "ponerle" el CD.

   Escoja **Configuración** y después **Almacenamiento** para agregar un CD y poner la imagen del SO en él. Observe que la máquina tiene el disco duro que se hizo en la guía anterior (DiscoNM.vdi) y que no tiene CD. Escoja el botón que tiene el dibujo de un CD con la etiqueta **Vacío** (figura A.10).

71 Complementary Metal-Oxide-Semiconductor (CMOS). Chip de memoria alimentado con batería que almacena la información para el proceso de inicio de la computadora. El sistema de entrada/salida básico (Basic Input Output System – BIOS) del equipo utiliza esta información cada vez que se enciende el equipo.

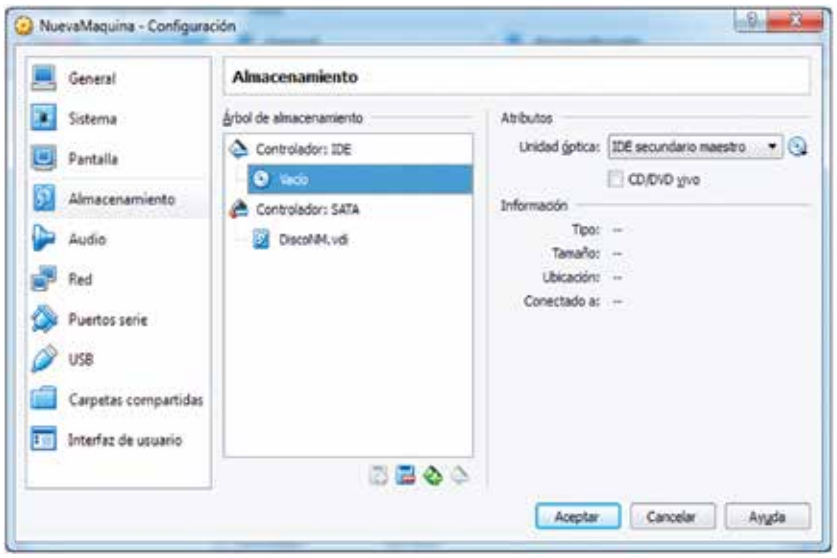


Figura A.10. Agregando un CD.

Después, localice el lugar donde está el ISO del SO que se desee instalar. Use el botón en forma de disco que está a la derecha de la etiqueta IDE **secundario maestro** (figura A.10), navegue por el sistema de archivo de la máquina hospedera y agregue el archivo con la imagen de instalación del SO.

Observe la figura A.11, que representa el momento en que se ha seleccionado el archivo imagen (.ISO) que contiene la instalación del SO (Debian en este caso). La figura es prácticamente igual a la A.10, pero ahora la máquina tiene una "disquetera de CD" en la que se encuentra "insertado" un "CD" con la instalación del SO Debian.

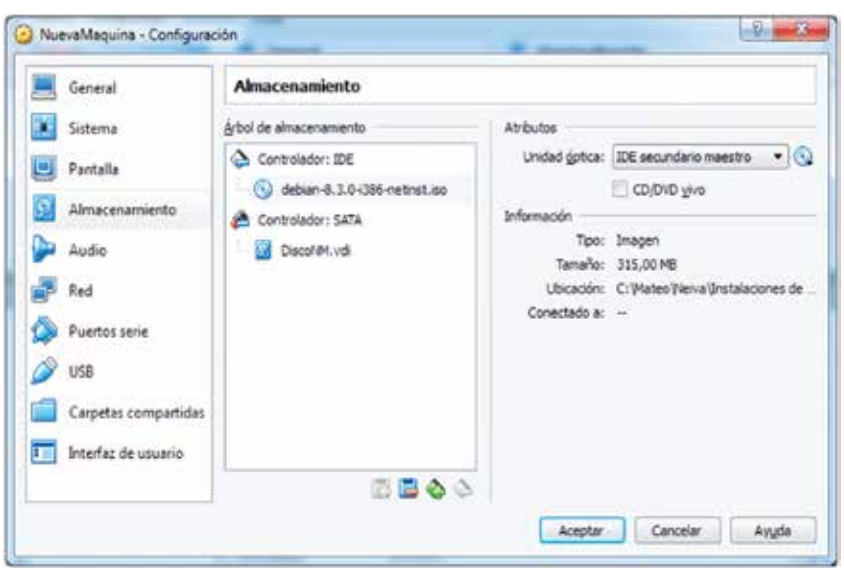


Figura A.11. CD insertado en la "disquetera".

Presione el botón **Aceptar**.

Ya todo está listo para comenzar el proceso de instalación del SO. De aquí en adelante, los pasos son específicos del SO que se instale y en nada difieren de la instalación del SO en una máquina real.

A.3 GUÍA DE INSTALACIÓN DEL SO DEBIAN

Las guías A1 y A2 son generales, la guía A3 se refiere específicamente al SO Debian

1. Seleccione la máquina que ha creado antes (pudiera haber otras). Presione el botón **Iniciar** del VirtualBox para comenzar el proceso de instalación (se supone que ya hizo las dos guías anteriores).

Figura A.12. Iniciando el proceso de instalación del SO Debian.

Cuando se presente una imagen como la de la figura A.12, presione la tecla de retorno para iniciar el proceso de instalación con la opción por defecto. Como ejercicio posterior, deberá practicar diversas formas de instalar el SO. Observe que gracias al uso de las máquinas virtuales eso no afectará a su equipo, de modo que podrá practicar cuanto quiera.

2. Se escogerá el idioma español para usarlo durante el proceso de instalación, con el objetivo de facilitar la comunicación con todos los lectores de habla hispana (figura A.13).

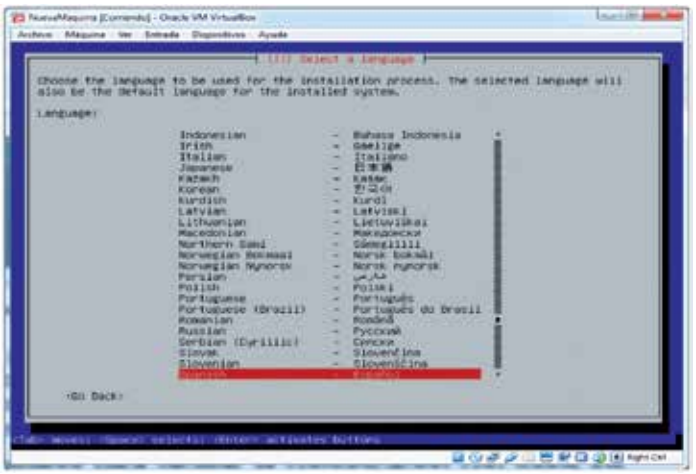


Figura A.13. Seleccionando el idioma para el proceso de instalación.

Una vez seleccionado el idioma, presione la tecla de retorno

3. Seleccione el país (figura A.14). Deberá leer todos los mensajes que le envíen los instaladores antes de tomar cualquier decisión; si no lo hace, puede ser que haga malas selecciones.

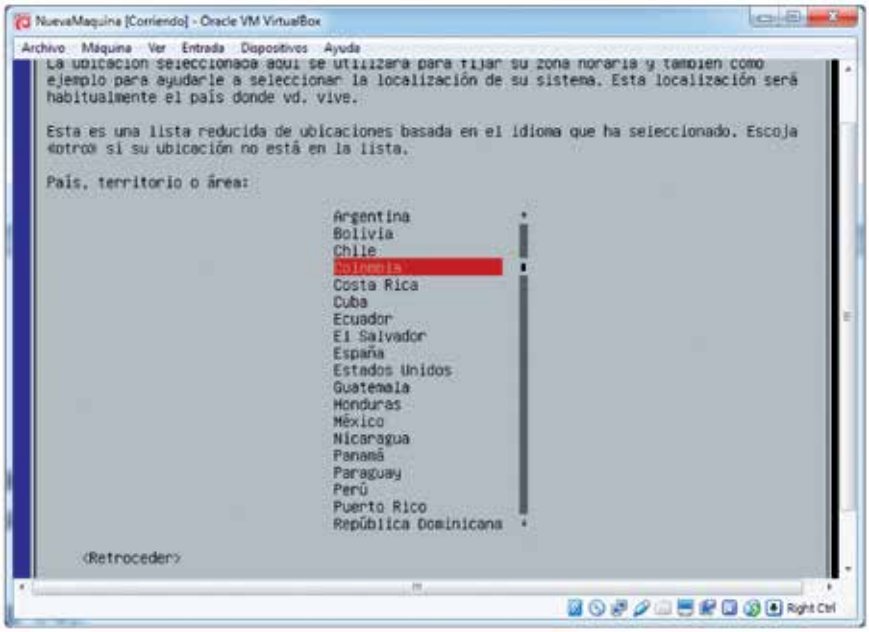


Figura A.14. Seleccionando el país.

Presione la tecla de retorno.

4. Ahora deberá escoger el teclado. Antes de hacerlo, cerciórese del teclado físico que usted está usando.

Si hace una mala selección, las teclas quedarán en lugares diferentes a los que se indican en el teclado; por ejemplo, no se localiza la **ñ** u otros símbolos, como la **@**, porque están en otros lugares, lo cual tiende a confundir.

Dicho problema se puede solucionar después, eligiendo la configuración apropiada, pero este es buen ejemplo de la toma de decisiones no acertadas motivadas por no leer bien los avisos de los instaladores.

Seleccione el teclado de acuerdo con su equipo físico y presione la tecla de retorno (observe la figura A.15).

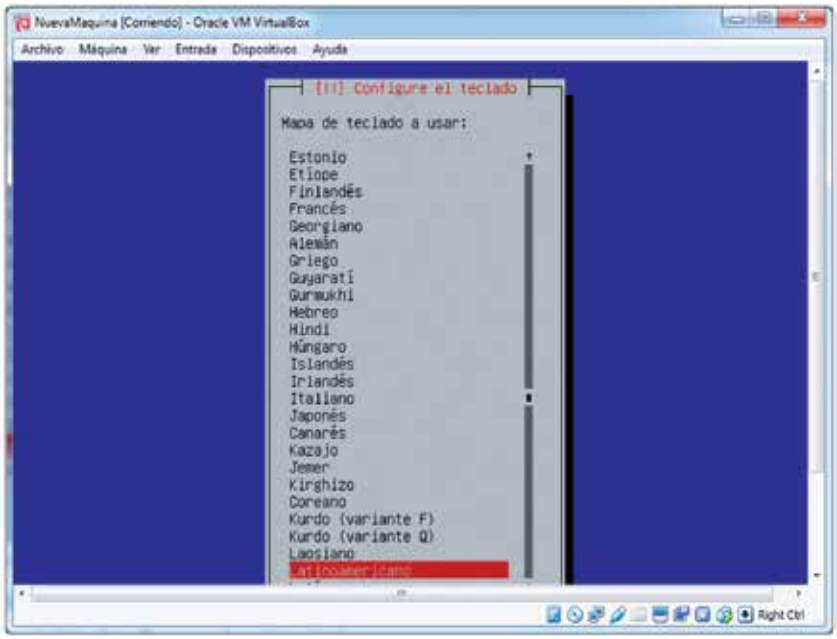


Figura A.15. Configurando el teclado.

5. Deberá establecerse un nombre para la máquina. Se escogió MiDebian en este caso (figura A.16), pero el nombre posiblemente deberá seguir ciertas especificaciones si la máquina forma parte de una red.

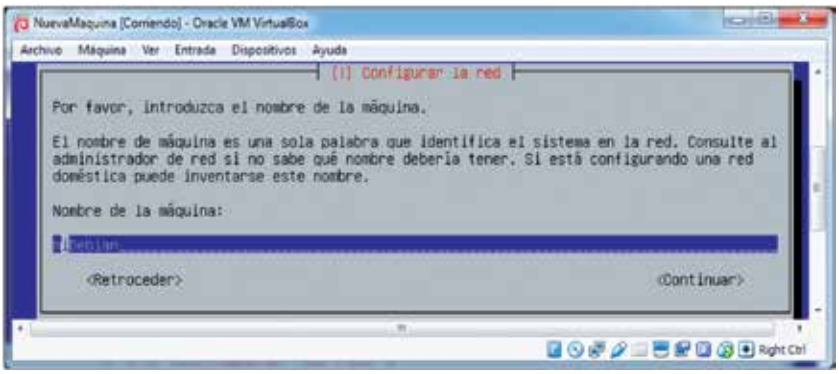


Figura A.16. Estableciendo el nombre de la máquina.

Póngale el nombre que desee y presione la tecla de retorno.

6. Después deberá especificarse el nombre del dominio. Este libro se escribió en la Universidad Cooperativa de Colombia y por eso el nombre del dominio es **ucc.edu.co** (figura A.17). Si su máquina no va a formar parte de un dominio específico, puede poner cualquier otro nombre.

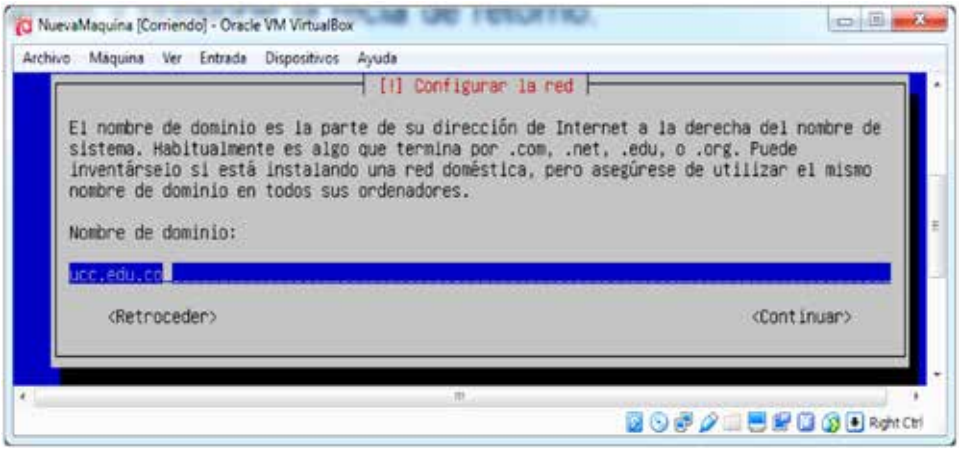


Figura A.17. Estableciendo el nombre del dominio.

Fije el nombre del dominio y presione la tecla de retorno.

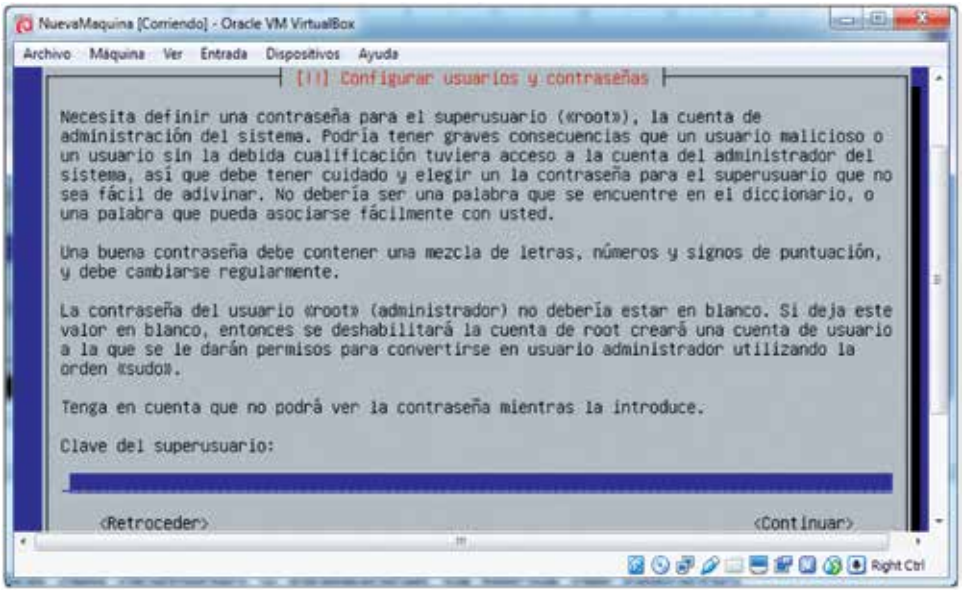


Figura A.18. Fijando la clave del superusuario (*root*).

7. Ahora deberá establecer una palabra clave para el superusuario (figura A.18). En un proceso de instalación, la clave que se escoja no debe ser muy complicada,

porque puede olvidarse. Por ahora, puede ser algo sencillo que después deberá cambiar por otra clave que posea la fortaleza necesaria. El sistema le pedirá la palabra clave dos veces para asegurarse de que la ha tecleado bien.

8. En este paso, el sistema le pide su nombre (debería ser el nombre completo, pero no es obligatorio). Ese nombre se asociará a una cuenta alternativa que se creará para que la use cuando no esté realizando funciones de superusuario (figura A.19). Este usuario también necesita un nombre (*login*) para poder entrar a la máquina y una palabra clave (figura A.20).

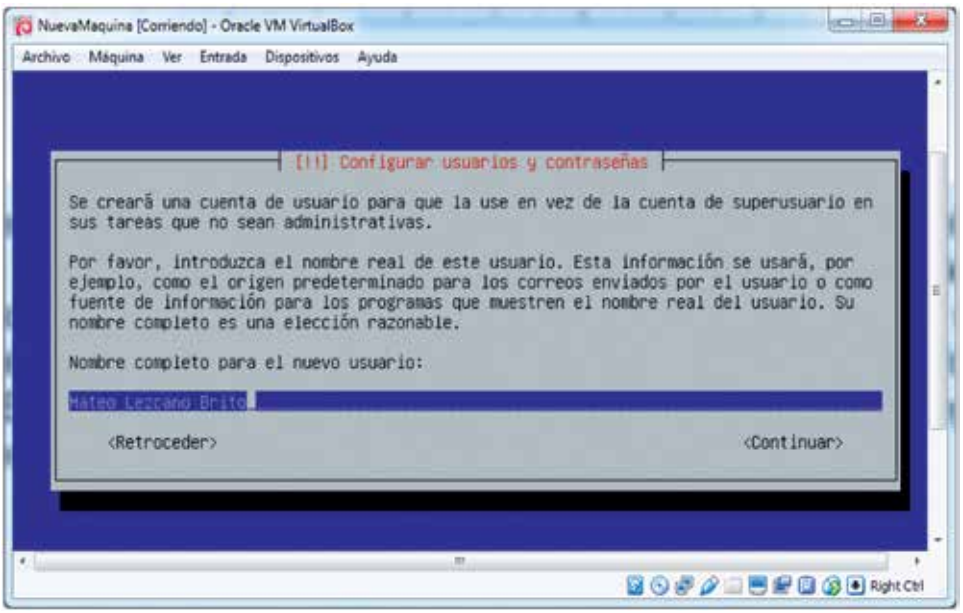


Figura A.19. Nombre completo del usuario para una cuenta alternativa.

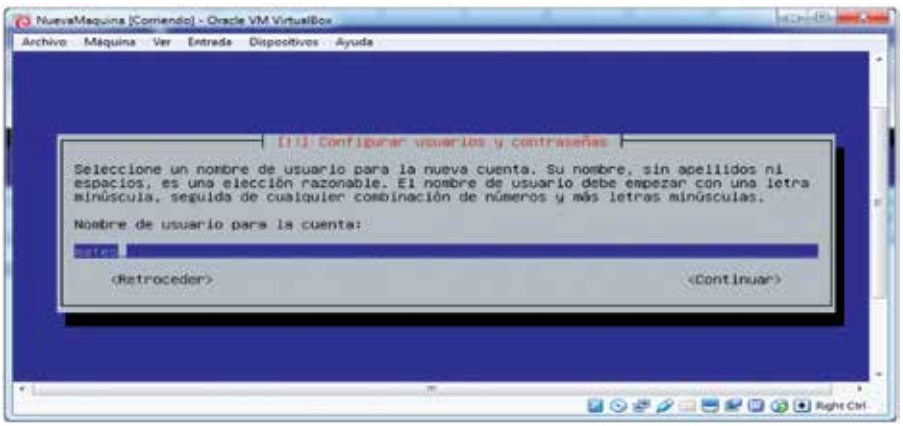


Figura A.20. Nombre de usuario (*login*) para la cuenta alternativa.

9. Después de lo anterior, comienza el proceso para particionar el disco duro (figura A.21), que puede ser guiado (con tres alternativas) o manual. En este caso y por

razones de simplicidad, se escoge el primer método de los cuatro propuestos. Escoja la primera opción y oprima la tecla de retorno.

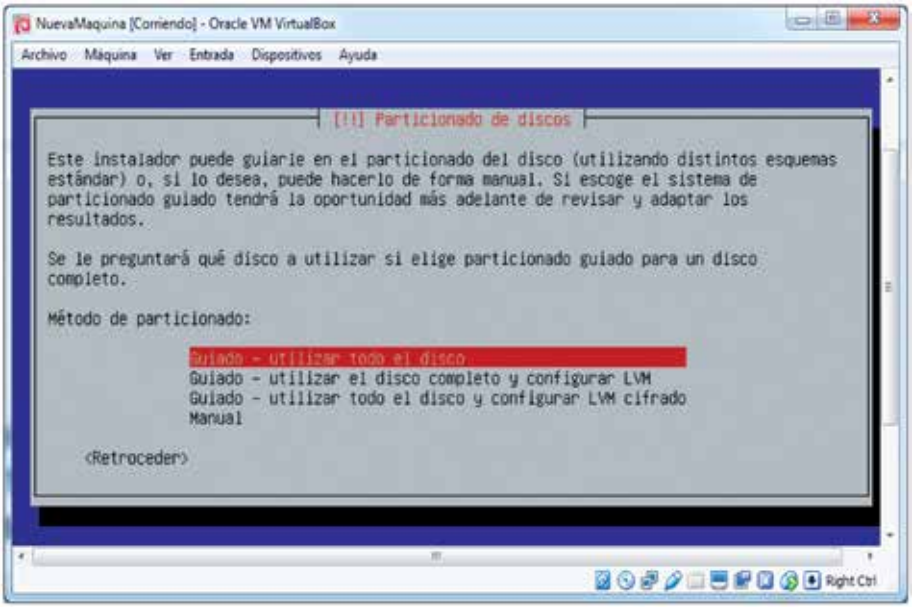


Figura A.21. Elegiendo el método de particionamiento.

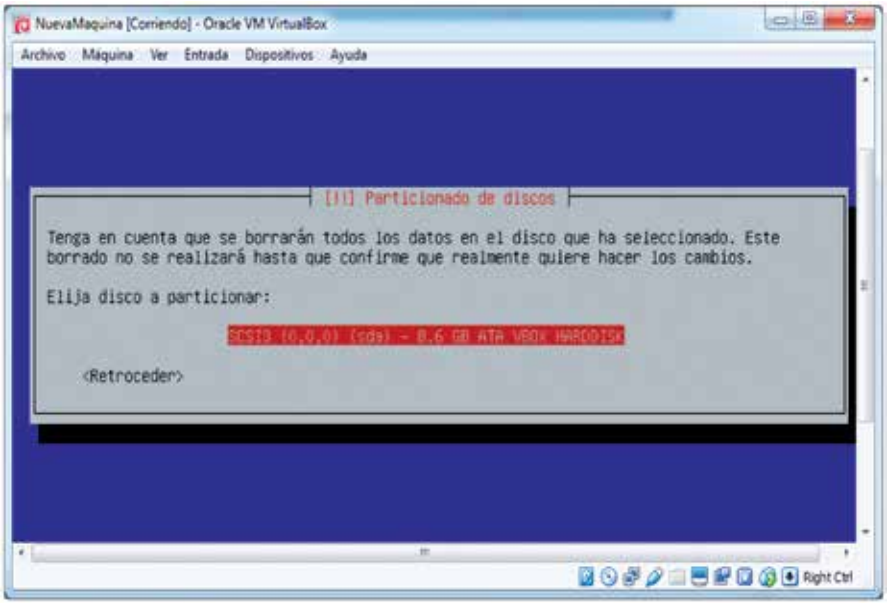


Figura A.22. Eligiendo el disco a particionar.

10. En este caso, hay un solo disco para particionar, de modo que se oprime la tecla de retorno (figura A.22).

11. Existen tres alternativas para hacer las particiones (figura A.23). Se recomienda la primera, debido a que las restantes necesitan conocimientos acerca de las particiones y su montaje, lo cual puede ser un ejercicio de estudio posterior.

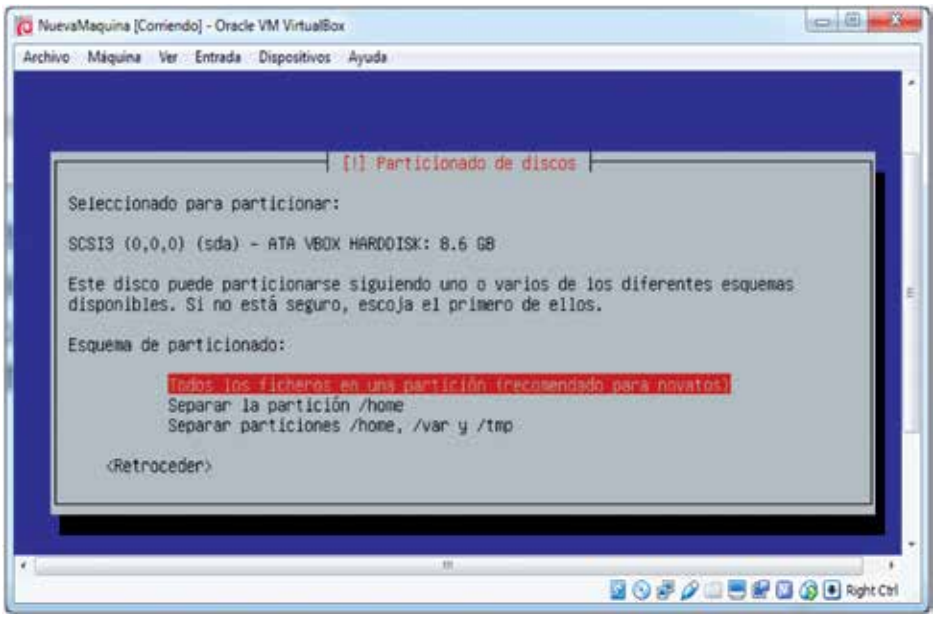


Figura A.23. Eligiendo el esquema para particionar.

Escoja la primera alternativa y oprima la tecla de retorno.

En este punto, ya todo está listo para comenzar a hacer las particiones. El sistema presenta un resumen (figura A.24) con todos los detalles acerca de lo que se hará. Si hay algún parámetro que no es el deseado, este es el momento de retroceder y arreglarlo.

Se especifica que se configura:

* RAID[72] por *software*.
* LVM[73].
* Los volúmenes de cifrado.
* Los volúmenes ISCSI[74].

Observe, además, que se hacen dos particiones:

* La primera fungirá como partición primaria, tiene una capacidad de 8.2 GB y se formateará con el sistema de archivos ext4.

72 **R**edundant **A**rray of Independent **D**isks. Técnica que proporciona una vía para almacenar los mismos datos en diferentes lugares (redundancia), sobre múltiples discos.

73 **L**ogical **V**olume **M**anager. Administrador de volúmenes lógicos para el *kernel* de Linux.

74 **I**nternet **S**mall **C**omputer **S**ystems **I**nterface. Estándar que permite usar el protocolo SCSI sobre redes TCP/IP.

* La segunda la usará el módulo de administración de la memoria como partición de intercambio para la memoria virtual.

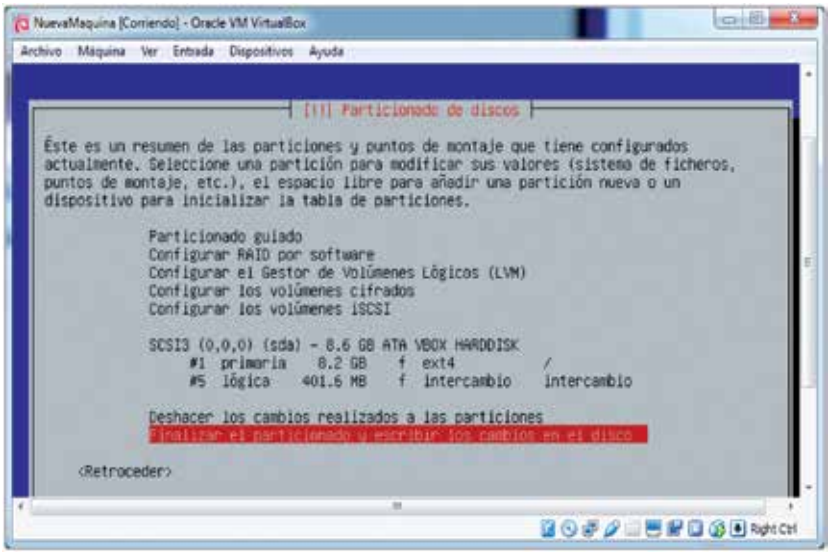


Figura A.24. Resumen del proceso de particionado que se llevará a cabo.

Oprima la tecla de retorno.

12. Después de haber leído los detalles del proceso de partición y del ulterior formateo de los discos involucrados, el usuario deberá elegir si desea continuar o si hay algo que debe arreglar antes, dado que este es un proceso irreversible (observe la figura A.25).

Figura A.25. Confirmando que se hará el proceso de partición.

Escoja la opción **Sí** y presione la tecla de retorno.

Después de esta acción, comienza el proceso de particionar los discos para después formatearlos con los sistemas de archivos escogidos e instalar los elementos básicos del SO. El proceso puede tomarse un tiempo considerable.

13. Una vez culminado todo el proceso anterior, el SO pide que se fije el país donde radica el repositorio que se usará por defecto[75].

Figura A.26. Fijando el repositorio.

Observe la figura A.26 y escoja el que considere más apropiado.

Presione la tecla de retorno.

14. Escoja una de las opciones posibles para configurar el gestor de paquetes, tomando como base algo como lo mostrado en la figura A.27.

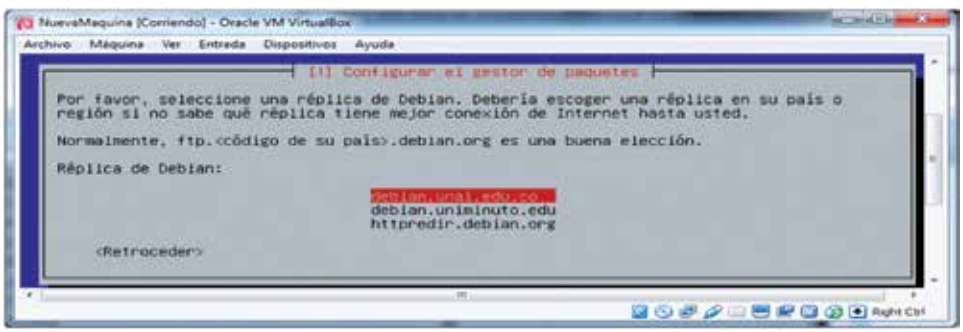


Figura A.27. Configurando el gestor de paquetes.

15. Configure el *proxy* si es necesario. En este caso, se dejará en blanco (figura A.28).

75 El repositorio es un lugar donde se pueden encontrar las actualizaciones del SO y muchos programas complementarios. Se puede cambiar editando el archivo /etc/apt/sources.list.

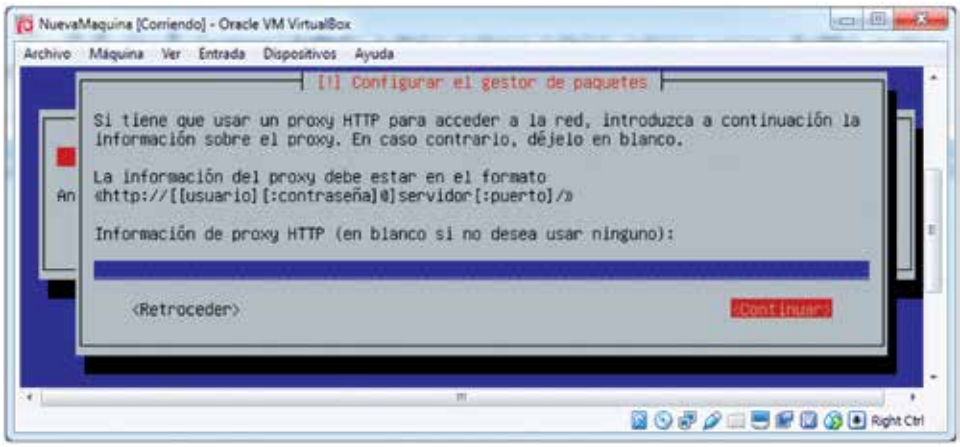


Figura A.28. No se usará *proxy*.

El sistema comienza a hacer un análisis de la réplica. Es posible que falle por varias razones, entre ellas: la desconexión de la máquina virtual de la red en la que está la máquina hospedera, que la réplica no esté disponible en ese momento, etc. En ese caso (sucedió cuando se estaba haciendo esta guía), se deberá retornar y escoger otro repositorio.

16. Una vez que se haya establecido correctamente el sitio de la réplica y se tenga acceso a él, el sistema comienza un proceso para descargar e instalar los archivos que necesite (puede tardar un tiempo considerable).

    Como parte de esta etapa, los desarrolladores del sistema le proponen participar en una encuesta automática para hacer un estudio acerca de los paquetes que más se usan. Observe la figura A.29.

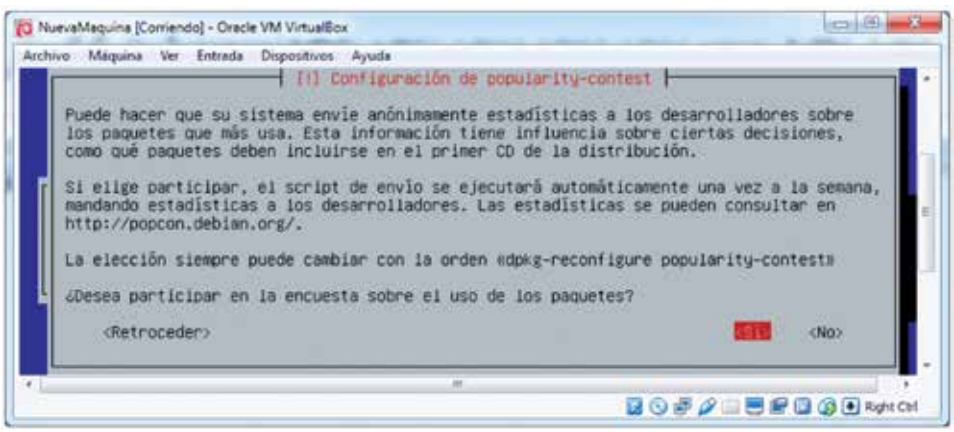


Figura A.29. Se participa en la encuesta.

Haga su elección y oprima la tecla de retorno.

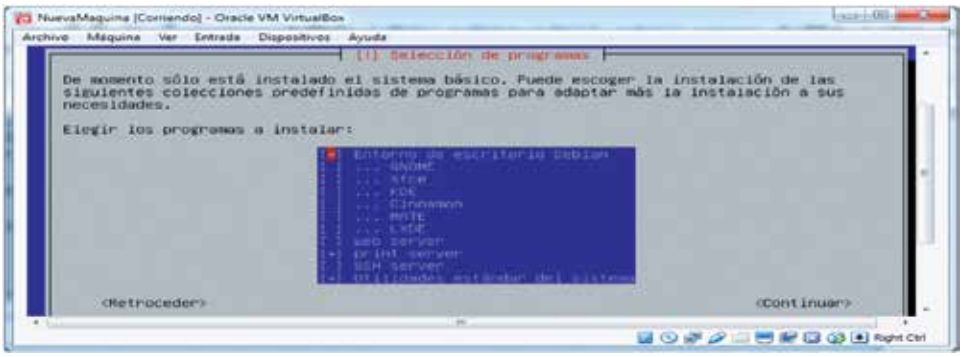


Figura A.30. Agregando nuevas colecciones.

17. En esta etapa, se pueden escoger las colecciones que se desean agregar al sistema (figura A.30). Escójalas y oprima la tecla de retorno, el proceso también puede ser demorado (depende del repositorio y de la cantidad de paquetes que se deseen agregar).

18. Ahora debe escogerse el cargador de arranque que se usará. La instalación propone el GRUB[76], deje esa opción y oprima la tecla de retorno (observe la figura A.31).

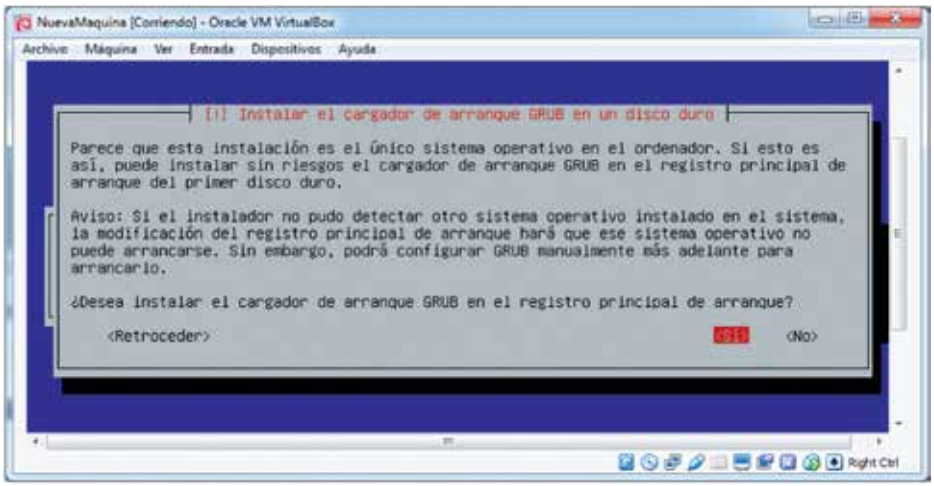


Figura A.31. Definiendo el cargador que se usará.

19. Después de haber decidido cuál es el cargador que se instalará, se permite elegir el lugar de instalación (figura A.32). La mayoría de los instaladores de SO toma esa decisión por sí mismo y, por defecto, el instalador se instala en el MBR[77] del primer disco duro. Marque la segunda opción y oprima la tecla de retorno.

76 GNU G Rand Unified Boot loader. Gestor de arranque múltiple de GNU.

77 **M**aster **B**oot **R**ecord. Primer sector de un dispositivo de almacenamiento.

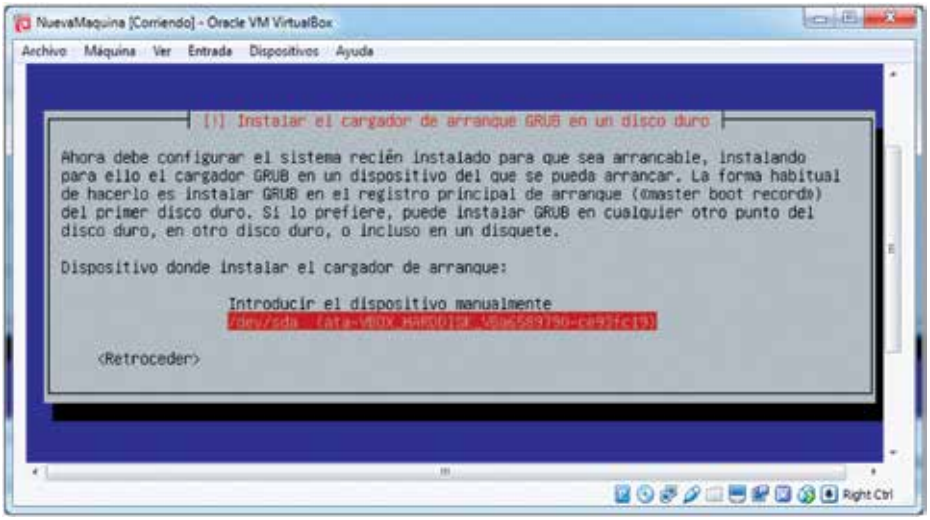


Figura A.32. Definiendo el lugar de instalación del cargador.

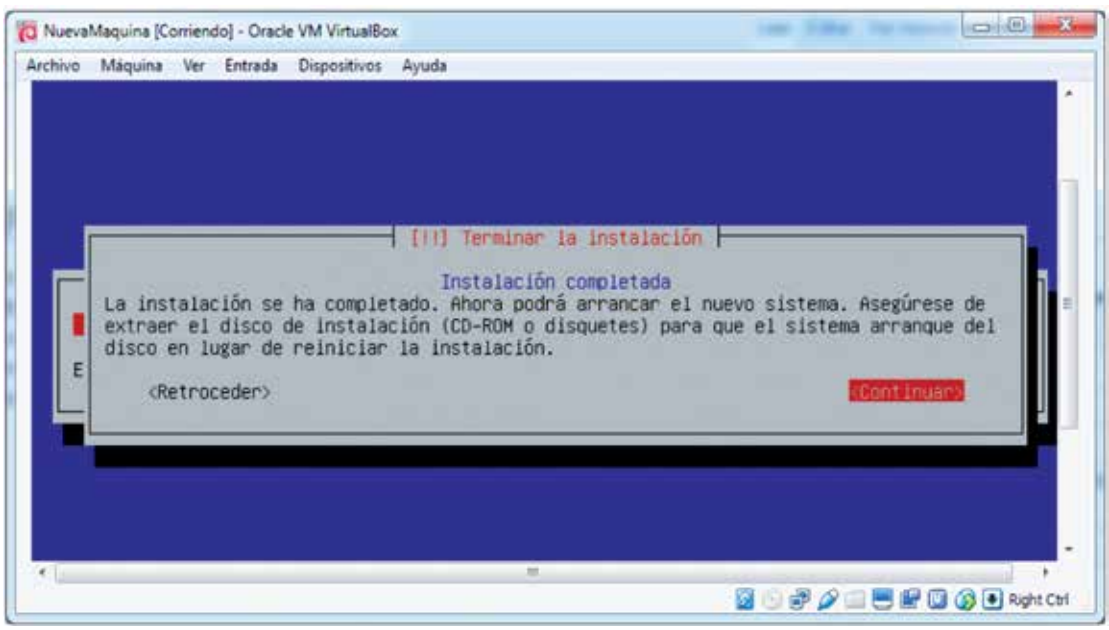


Figura A.33. Fin del proceso de instalación.

20. Ya está terminado el proceso de instalación (figura A.33). La figura recomienda quitar el CD de instalación antes de iniciar la máquina, a fin de que el SO no vuelva a intentar el arranque desde ese dispositivo (en ese caso, comenzaría de nuevo el proceso de instalación). Esa acción física no se puede hacer porque no se tiene un CD real sino virtual.

    Oprima la tecla de retorno.

En el caso que se está analizando (SO instalado sobre una máquina virtual), al oprimir la tecla de retorno, el sistema deberá iniciarse normalmente mostrando el nombre completo de usuario que usted tecleó en el paso 8.

Oprima de nuevo la tecla de retorno y escriba su contraseña. Este paso se resume en la figura A.34.

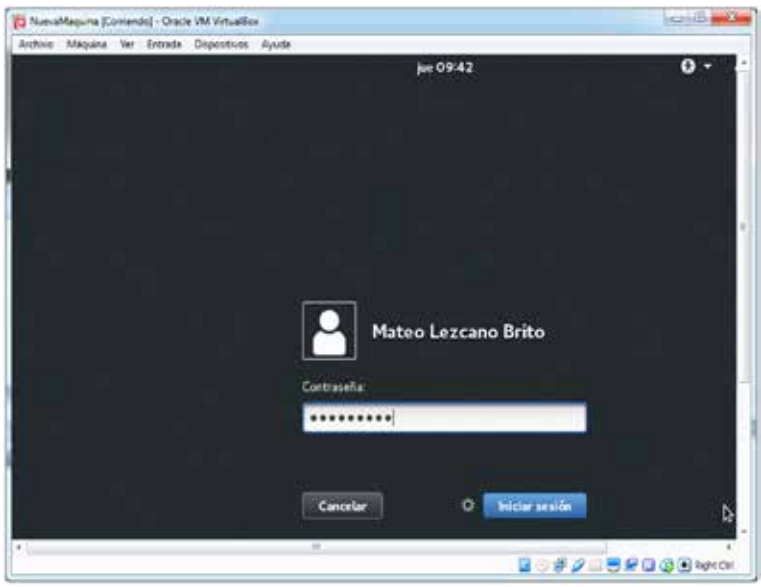


Figura A.34. Entrando al sistema.

21. La figura A.35 muestra el resultado visible del trabajo, mientras que la figura A.36 muestra el resultado no visible:

    El SO Debian se ha instalado sobre la máquina virtual MiMaquina que está soportada sobre VirtualBox, el cual a la vez está soportado sobre el SO Windows 10 que está instalado sobre una máquina real.

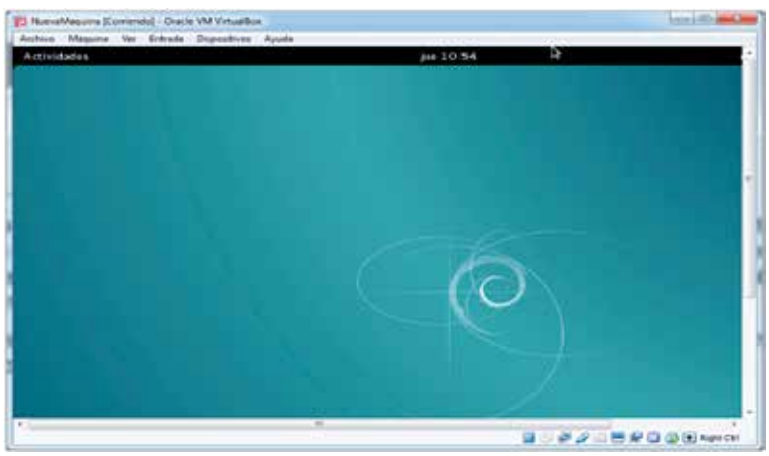


Figura A.35. SO Debian en espera de órdenes.

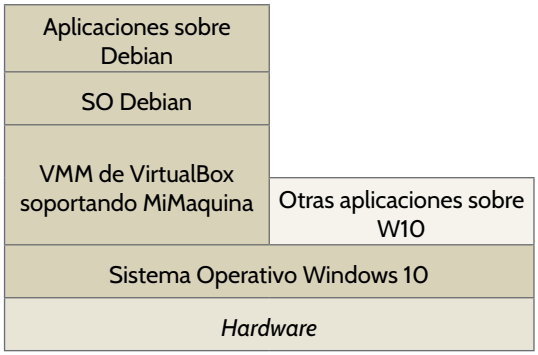


Figura A.36. Idea esquematizada de las capas usadas en la virtualización.

22. Salga del SO Debian para que se apague la máquina y vaya a **Configuración | Sistema** del VirtualBox, a fin de dejar el disco duro como única opción en el orden de arranque (figura A.37).

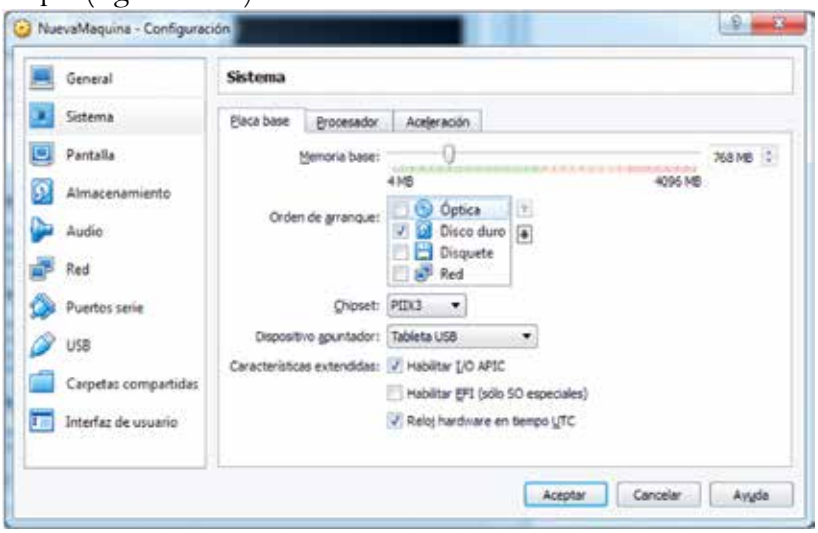


Figura A.37. Estableciendo el disco de arranque.

Oprima el botón **Aceptar** e inicie la máquina.

Ahora se ha terminado el trabajo, el disco que se ha creado se puede usar en otras máquinas y así conservar el trabajo.

Sugerencia para los profesores:

Las prácticas relativas a los SO tipo Unix que se proponen en el presente libro pueden hacerse con la máquina virtual que se ha creado.

Los estudiantes deberían entregar sus archivos virtuales (*.vdi) para que el profesor pueda revisar los ejercicios. Recomiende a los estudiantes que creen dos directorios:

* Uno para los programas hechos en C.
* Otro con los programas en Shell script.

Sugerencia para los estudiantes:

Mantenga sus directorios organizados; todos los archivos que cree póngalos en un directorio con nombres adecuados que sean hijos de /home/<nombre de usuario>.

Los nombres de los archivos deben seguir una convención que haga fácil deducir su contenido.

Recuerde que Unix es sensible al tipo de letra; los archivos pueden tener un nombre que combine dos palabras: la primera debería estar en minúscula sostenida y la segunda debería comenzar con letra mayúscula, por ejemplo: esteArchivo.

Tenga varias copias de su disco virtual en diferentes lugares.

Índice analítico

Este libro busca aportar herramientas de gestión y análisis de los sistemas operativos para estudiantes de ingeniería de sistemas. Para ello, expone fundamentos teóricos alrededor de cuestiones como: cuáles son las formas de planificación del procesador central y qué algoritmos se deben usar para distribuir el tiempo de procesamiento; qué es un archivo como concepto informático y cómo se organiza la información; y cuáles son las técnicas más eficientes para la administración de memoria. Además, para ilustrar de manera práctica estos conceptos, el libro presenta un caso de estudio basado en el análisis del sistema operativo Unix y sus descendientes. Por último, con el fin de completar la panorámica de este trabajo, el autor deja a disposición del lector un instructivo para la instalación de máquinas virtuales, las cuales son esenciales para la experimentación en esta área de conocimiento.

ISBN 978-958-760-107-7

Universidad Cooperativa de Colombia