UNIX: Una puerta abierta a los sistemas abiertos

Santiago Garcés Guevara
TABLA DE CONTENIDO

CAPÍTULO 1
SISTEMA OPERATIVO UNIX................................................................. 1
  1.1 PARTES DEL SISTEMA OPERATIVO UNIX................................. 2
  1.2 INGRESO Y SALIDA DEL SISTEMA UNIX................................. 2
    1.2.1 INGRESO AL SISTEMA .................................................. 3
    1.2.2 SALIDA DEL SISTEMA .................................................. 4
  1.3 CONFIGURACIÓN DE TECLAS DEL TERMINAL......................... 4
    1.3.1 EL CARACTER DE BORRADO DE CARACTERES .................... 4
    1.3.2 EL CARACTER DE ELIMINACIÓN DE LINEA ......................... 4
    1.3.3 CAMBIO DE LOS CARACTERES DE BORRADO Y ELIMINACION ...... 5
  1.4 ORGANIZACIÓN DEL SISTEMA DE FICHEROS (FILE SYSTEM).......... 5
    1.4.1 FICHEROS .................................................................... 5
      1.4.1.1 Ficheros regulares.................................................. 5
      1.4.1.2 Ficheros especiales.................................................. 6
    1.4.2 DIRECTORIOS ............................................................... 6
    1.4.3 SISTEMA DE FICHEROS .................................................. 7
    1.4.4 CONVENCIONES PARA NOMBRAR A UN DIRECTORIO O FICHERO 8
      1.4.4.1 Nombres de ficheros ............................................... 8
      1.4.4.2 Ruta o camino de un fichero ..................................... 8
      1.4.4.3 Uso de metacaracteres ............................................. 9
  1.5 COMANDOS .......................................................................... 10
    1.5.1 SINTAXIS DE UN COMANDO ............................................ 10
    1.5.2 REDIRECCIONAMIENTO .................................................. 11
    1.5.3 RESUMEN DE COMANDOS ................................................ 11
      1.5.3.1 Manipulación de ficheros ........................................... 11
        1.5.3.1.1 Listado de ficheros.......................................... 11
        1.5.3.1.2 Despliegue de ficheros ..................................... 12
        1.5.3.1.3 Copiado de ficheros ......................................... 13
        1.5.3.1.4 Renombre de ficheros ....................................... 13
        1.5.3.1.5 Borrado de ficheros .......................................... 13
        1.5.3.1.6 Cambio de permisos en los ficheros .................... 13
      1.5.3.2 Manipulación de directorios ....................................... 14
        1.5.3.2.1 Creación de directorios ..................................... 14
        1.5.3.2.2 Borrado de directorios ..................................... 15
        1.5.3.2.3 Cambio de directorio ......................................... 15
        1.5.3.2.4 Impresión del directorio actual ........................... 15
      1.5.3.3 Comunicación con otros usuarios .................................. 16
        1.5.3.3.1 Envío de correo electrónico ................................ 16
        1.5.3.3.2 Envío de mensajes a una terminal ....................... 17
      1.5.3.4 Otros comandos de UNIX ............................................ 17
  1.6 EDITOR DE TEXTOS VISUAL (VI) .............................................. 19
# CAPITULO 2

** LENGUAJE DE PROGRAMACION SHELL ................................................................. 23**

2.1 COMANDOS BACKGROUND ............................................................................. 23
2.2 REDIRECCION DE ENTRADA/SALIDA ............................................................ 24
2.3 PIPES (CONDUCTOS) Y FILTROS .................................................................. 24
2.4 EL LENGUAJE SHELL ....................................................................................... 25
   2.4.1 CREACION DE UN SHELL ......................................................................... 25
   2.4.2 EJECUCION DE FICHEROS SHELL ......................................................... 25
   2.4.3 VARIABLES SHELL .................................................................................. 26
      2.4.3.1 Mediante asignaciones ..................................................................... 26
      2.4.3.2 Mediante parametros posicionales ............................................... 26
      2.4.3.3 Mediante set .................................................................................... 27
      2.4.3.4 Mediante read ................................................................................. 27
   2.4.4 DESPLIEGUE DE SALIDA ........................................................................ 27
   2.4.5 COMANDOS USADOS EN UN SHELL ....................................................... 28
      2.4.5.1 Referencia a comandos de Unix ..................................................... 28
      2.4.5.2 Comando test ................................................................................. 28
      2.4.5.3 Estructuras de control .................................................................... 28
         2.4.5.3.1 Sentencia CASE ....................................................................... 29
         2.4.5.3.2 Sentencia FOR ........................................................................... 29
         2.4.5.3.3 Sentencia IF .............................................................................. 30
         2.4.5.3.4 Sentencia WHILE ..................................................................... 30
         2.4.5.3.5 Sentencia UNTIL ....................................................................... 30
      2.4.5.4 Comandos BREAK y CONTINUE ...................................................... 30
      2.4.5.5 Comando EXPR ............................................................................... 31
      2.4.5.6 Comando ECHO .............................................................................. 31
      2.4.5.7 Comando SHIFT .............................................................................. 31
      2.4.5.8 Comando TRAP ............................................................................... 32
      2.4.5.9 Comando EXIT ............................................................................... 32
   2.4.6 OPERADORES ESPECIALES .................................................................... 33
   2.4.7 VARIABLES ESPECIALES EN SHELL ..................................................... 33
   2.4.8 VARIABLES SHELL .................................................................................. 33
2.5 GRAMATICA DE SHELL ................................................................................. 34
2.6 META CARACTERES Y PALABRAS RESERVADAS .......................................... 35
2.7 PROGRAMAS EJEMPLO ................................................................................... 36

# CAPITULO 3

** LENGUAJE DE PROGRAMACIÓN AWK .............................................................. 39**

3.1 ESTRUCTURA DEL PROGRAMA ..................................................................... 39
3.2 CONVENCION LEXICA ................................................................................ 40
CAPITULO 5

LLAMADAS AL SISTEMA UNIX

5.1 ARQUITECTURA DEL SO UNIX ..................................................74
  5.1.1 SUBSISTEMA DE FICHEROS ............................................75
  5.1.2 SUBSISTEMA DE DISPOSITIVOS .......................................75
  5.1.3 SUBSISTEMA DE CONTROL DE PROCESOS ......................75
  5.1.4 CONTROL DE HARDWARE ................................................75
5.2 LLAMADAS AL SISTEMA ..........................................................76
  5.2.1 DESCRIPCION DE ALGUNAS LLAMADAS AL SISTEMA ........77
    5.2.1.1 OPEN ..............................................................77
    5.2.1.2 CLOSE ..........................................................80
    5.2.1.3 READ ..........................................................81
    5.2.1.4 WRITE ..........................................................82
    5.2.1.5 CREATE ........................................................84
    5.2.1.6 PIPE ............................................................85
    5.2.1.7 DUP ..............................................................85
    5.2.1.8 CHOWN ........................................................86
    5.2.1.9 CHMODE .........................................................87
  5.2.2 EJERCICIO PRACTICO .......................................................88
    5.2.2.1 Listado del Programa .............................................88
    5.2.2.2 Descripción del programa .....................................89

CAPITULO 6

ADMINISTRACION DEL SISTEMA .....................................................91

6.1 INICIACION Y APAGADO DEL EQUIPO .....................................92
  6.1.1 INICIACION DEL EQUIPO ................................................92
  6.1.2 APAGADO DEL EQUIPO ....................................................92
6.2 ADMINISTRACION DE USUARIOS ..............................................93
6.2.1 EL FICHERO DE PASSWORDS 
6.2.2 EL FICHERO DE GRUPOS 
6.3 ADMINISTRACION DEL SISTEMA DE FICHeros 
6.4 CONTABILIDAD DEL SISTEMA 
6.5 RESPALDOS Y RECUPERACION 
6.5.1 COPIA DE GRUPOS DE FICHEROS AISLADOS 
6.5.2 COPIA TOTAL DEL SISTEMA DE FICHEROS 
6.6 CONFIGURACION DE PERIFERICOS 
6.7 CONTROL DE PROCESOS EN EL TIEMPO 

BIBLIOGRAFIA
Un sistema operativo es un conjunto de programas cuyo propósito es organizar y controlar eficientemente los recursos de un ordenador, de forma que éste pueda ser empleado por usuarios finales (no especialistas). Estos recursos incluyen memoria, disco, impresoras, terminales, dispositivos periféricos adicionales conectados al sistema y programas de aplicación.

El sistema Unix consiste de un sistema operativo interactivo, multiusuario, multitarea, y más de doscientos programas de aplicación y utilitarios. Un sistema interactivo implica interacción entre usuario y ordenador. Esta interacción se produce cuando el usuario le da órdenes al ordenador que éste le responde inmediatamente con los resultados de la ejecución de tales órdenes. Esta interacción continua mientras el usuario esté en sesión de trabajo con el ordenador. Un sistema operativo multitarea significa que el sistema puede realizar varias tareas -llamadas procesos- simultáneamente. Esto implica que, al mismo tiempo, se puede dar al sistema una o más tareas para que las ejecute, sin necesidad de esperar a que se desocupe de una tarea para indicarle que haga otra.

La característica multiusuario significa que el sistema puede atender a más de una persona al mismo tiempo, dando la impresión al usuario que el ordenador le atiende únicamente a él. El aspecto multiusuario es una consecuencia natural de la característica multitarea descrita anteriormente, que hace posible que grupos de personas puedan trabajar juntas fácilmente, compartiendo información y facilidades comunes a través del sistema operativo.
1.1 PARTES DEL SISTEMA OPERATIVO UNIX

El Sistema Unix se compone de tres partes principales: herramientas y utilitarios, shell y el Kernel.

Usuario

Herramientas y Utilidades

SHELL

KERNEL

Medio de programación

a) Herramientas y Utilitarios

Son los comandos (órdenes para el sistema operativo) y los programas de aplicación que se pueden ejecutar en la máquina.

b) Shell

Es el intérprete de las órdenes que el usuario presenta al ordenador. Cuando se ingresa un requerimiento al sistema (por ejemplo ejecutar un comando o una aplicación), shell interpreta lo que el usuario requiere y hace que el Kernel lo ejecute en la circuitería de la máquina. El shell además de ser un intérprete de comandos es también un lenguaje de programación. A un programa shell que consta de un conjunto de comandos se lo llama shell scripts.

c) Núcleo (Kernel)

El Kernel o núcleo es la parte central del sistema operativo. Administra los recursos del ordenador, ocupándose de la memoria, los discos, cintas, impresoras, terminales, líneas de comunicación y cualquier otro dispositivo.

Todas estas tareas son imperceptibles por el usuario, permitiéndole enfocar su atención en su trabajo y no en el ordenador.

1.2 INGRESO Y SALIDA DEL SISTEMA UNIX

En esta sección se describe la manera de "ingresar" y "salir" del sistema operativo UNIX.
1.2.1 **INGRESO AL SISTEMA**

Para poder accesar al sistema Unix es necesario que el usuario esté registrado debidamente en el sistema. Para ello, debe tener asociado consigo un nombre de usuario (login) que lo identifique y una palabra secreta o contraseña (password) para seguridad.

Primeramente la terminal del usuario tiene que estar conectada por cable al sistema en cuestión. El usuario puede darse cuenta que está su terminal conectada al sistema al observar en la pantalla un mensaje como el siguiente:

```
login:
```

Para iniciar la sesión, el usuario simplemente escribe el respectivo nombre de usuario y, a continuación, pulsa la tecla de retorno de carro (etiquetada RETURN, ENTER o INTRO).

Una vez que ha ingresado el nombre de usuario, el sistema responde pidiendo su contraseña:

```
password:
```

El usuario escribe la contraseña y a continuación pulsa la tecla RETURN. Mientras el usuario escribe la contraseña, no se refleja la escritura de dicha contraseña en la pantalla, esto es debido a que la contraseña es privada, intransferible, personal y cualquier persona que esté cerca del lugar no podría verla. En caso de que el usuario olvide la contraseña tiene que pedir ayuda a la persona encargada de administrar el sistema.

Cuando el usuario ha escrito el nombre de usuario y contraseña, el sistema comprueba que realmente es un usuario conocido del sistema y que ha dado la contraseña correcta. Si por alguna razón, el inicio de sesión es incorrecto, sea que se pulsó incorrectamente el nombre de usuario o la contraseña, el sistema muestra el mensaje correspondiente:

```
Login incorrect
```

```
login:
```

En este caso el usuario tiene que volver a ingresar el nombre del usuario y la contraseña. Se debe tener presente que en Unix existe diferencia entre usar mayúsculas y minúsculas en un nombre.

Una vez que el usuario a digitado correctamente su nombre de usuario y contraseña, el sistema está listo para obedecer las órdenes que le presente el usuario. Como muestra de su buena voluntad y obediencia le presenta al usuario un prompt (una petición de orden. En la pantalla aparecerá el prompt de la siguiente manera:

```
$
```

Una vez que el sistema nos indica el prompt, podemos digitar un comando u orden a que ejecute el sistema. El signo $ aparecerá cada vez que el sistema ha terminado de ejecutar una orden y está listo para recibir la siguiente.
Al final de cada línea que el usuario escribe, las ordenes que le da a Unix, se debe que pulsar la tecla RETURN, para que el sistema tome en cuenta esa entrada. En este documento, no se mostrará explícitamente el RETURN al final de las líneas de los ejemplos.

### 1.2.2 SALIDA DEL SISTEMA

Para poder terminar una sesión de trabajo en Unix es necesario digitar:

```
$ exit <RETURN>
```

Después de esto el sistema presentará la indicación de login para que otro usuario pueda conectarse al sistema. Es decir, se presentará el siguiente mensaje:

```
login:
```

Otra manera de terminar la sesión es digitando las tecla Control y d, al mismo tiempo :

```
$ <ctrl-d>
```

### 1.3 CONFIGURACION DE TECLAS DEL TERMINAL

#### 1.3.1 EL CARACTER DE BORRADO DE CARACTERES

En la mayoría de las instalaciones del Sistema UNIX el carácter para borrar el carácter previo es el backspace. Está etiquetado como BACKSPACE o BS en algunos terminales. Otros terminales lo marcan como una flecha hacia la izquierda (control del cursor). En algunos terminales primitivos hay que pulsar <ctrl-h>, esto hace que el cursor se mueva un carácter a la izquierda en la pantalla. El carácter de borrar carácter le dice al sistema que ignore el último carácter que se escribió.

Cuando se utiliza el carácter # como carácter de borrado hay que contar el número de caracteres a corregir, algo que es innecesario cuando se usa el retroceso o <ctrl-h>.

Si el carácter de borrado es # y necesita introducir un # literalmente, tiene que decirle a UNIX que eso es realmente lo que quiere. Esto se consigue precediendo el # por un \. Así que para conseguir realmente el # tiene que pulsar \\.

La tecla \ es el carácter de "escape". Se utiliza para eliminar cualquier significado especial que el sistema pudiera asociar al carácter que le sigue.

#### 1.3.2 EL CARACTER DE ELIMINACION DE LINEA

El carácter de eliminación de línea al final de una línea significa "eliminar" o ignorar la línea escrita hasta ahora". Le dice a UNIX que el usuario no quiere que obedezca la línea de orden que acaba de escribir.

UNIX: Una puerta abierta a los sistemas abiertos
En muchos sistemas se ha dejado la @ como carácter de eliminación de línea, en otros se ha cambiado a un carácter de control. Dos alternativas conocidas son <ctrl-x> y <ctrl-u> como carácter de eliminación de línea.

No hay que pulsar <RETURN> después del carácter de eliminación. La forma de averiguar cuál es el carácter de eliminación, se lo realiza mediante el comando stty ejecutándolo desde el prompt:

```
$ stty <RETURN>
```

### 1.3.3 CAMBIO DE LOS CARACTERES DE BORRADO Y ELIMINACIÓN

Se puede cambiar los caracteres de borrado y de eliminación a cualquier otro que se quiera, utilizando la orden stty. Esto es especialmente útil si se está en un sistema donde los caracteres de borrado y eliminación son # y @.

Para cambiar los caracteres de borrado a <ctrl-h> se pondrá:

```
$ stty erase <ctrl-h>
```

Si se quiere cambiarlo de nuevo a #, se escribiría:

```
$ stty erase #
```

Para cambiar el carácter de eliminación a <ctrl-u>:

```
$ stty kill ^u
```

Ahora, cuando se presiona <ctrl-u>, el sistema ignora todo lo que haya en esa línea de orden.

### 1.4 ORGANIZACIÓN DEL SISTEMA DE FICHEROS (FILE SYSTEM)

UNIX tiene un sistema de ficheros donde los usuarios pueden almacenar y recuperar información en trozos con nombre llamados ficheros (files).

#### 1.4.1 FICHEROS

El fichero es la unidad fundamental del sistema Unix. Hay tres tipos diferentes de ficheros: ficheros regulares, directorios y ficheros especiales. Cada uno de estos tipos se describe a continuación.

##### 1.4.1.1 Ficheros regulares

Las ficheros regulares contienen típicamente información tal como documentos, datos o programas fuente. Los ficheros binarios ejecutables pertenecen también, a este tipo. Un fichero regular es simplemente una
concatenación de caracteres de 8 "bits" cada uno. Estos caracteres pueden ser interpretados por el ordenador como texto, instrucciones binarias, o sentencias de un lenguaje de programación de acuerdo al programa que los examine.

Todo fichero regular tiene los siguientes atributos:

- Un nombre de fichero (no necesariamente único)
- Un número único en el sistema llamado número de i-nodo
- Un tamaño en caracteres
- Una fecha de creación,
- Una fecha de última modificación
- Una fecha de último acceso
- Un grupo de permisos de aaaaaa

1.4.1.2 Ficheros especiales

Los ficheros especiales corresponden a los dispositivos físicos tales como la unidad de diskette, la unidad de disco, impresoras, terminales y la memoria del sistema. Se les denomina ficheros especiales de dispositivo.

1.4.2 DIRECTORIOS

Los directorios son ficheros de solo lectura que contienen información sobre los ficheros y subdirectorios que están en ellos. El comando ls se emplea para examinar directorios e imprimir información sobre los ficheros contenidos en el directorio.

El anidamiento de directorios dentro de otros directorios es la forma en que Unix implementa su sistema característico de estructura de árbol.

Con múltiples usuarios y múltiples proyectos, el número de ficheros en el sistema puede proliferar rápidamente. Afortunadamente Unix organiza todos los ficheros en una estructura jerárquica de árbol. Esta estructura puede concebirse como un mundo físico dentro del cual el usuario puede moverse de un lugar a otro lugar. Estos "lugares" son los directorios. Cada usuario del sistema tiene su propio directorio personal (HOME DIRECTORY). Dentro de ese directorio, el usuario puede tener subdirectorios directamente controlados por él.

Cuando se accede al sistema (login), el usuario queda colocado "en" su directorio (HOME DIRECTORY). A menos que se tome alguna acción especial cuando se crea un fichero, éste residirá en el directorio de trabajo. Dicho fichero no tendrá relación alguna con otros ficheros del mismo nombre residentes en directorios de otro usuarios.

Un diagrama típico de un directorio de usuario sería:

```
usr
```
En esta figura, el directorio usr contiene el directorio personal de cada usuario definido en el sistema. Note que el subdirectorio oficios en el directorio dnrs_t3 no tiene relación alguna con el subdirectorio oficios en el directorio dnrs_t2. Esto no tiene mucha importancia si los ficheros y subdirectorios de interés se hallan en el directorio de dnrs_t2, pero si dnrs_t2 y dnrs_3 trabajan juntos o si trabajan en proyectos distintos pero relacionados, esta división de ficheros se hace especialmente útil.

1.4.3 SISTEMA DE FICHeros

Un sistema de ficheros es un conjunto de ficheros organizados de alguna manera. En Unix, este conjunto de ficheros consiste de una estructura de árbol invertida. La siguiente ilustración muestra una estructura típica de un sistema de ficheros. En una estructura de sistema de ficheros, la "raíz" se encuentra a la cabeza y las "hojas" en la parte inferior. Los directorios corresponden a nodos del árbol. Si algún directorio contiene ramos hacia otros ficheros o directorios entonces éstos últimos se encuentran contenidos en el directorio. Es posible nombrar cualquier fichero del sistema comenzando desde la raíz y recorriendo todas las ramas hasta llegar al fichero dado (direccionamiento absoluto). De igual manera, es posible nombrar cualquier fichero del sistema en relación con cualquier directorio diferente del raíz (direccionamiento relativo).
En un sistema de ficheros de esta estructura, el árbol crece hacia abajo. Los nombres bin, usr, dev, dnr$s_t1$, dnr$s_t2$, dnr$s_t3$ representan directorios y son todos nodos del árbol. En Unix el nombre del directorio raíz se especifica mediante el carácter `/ (slash).

Los nombres texto1, texto2, oficio1, fact01, ofci003, ofci004 son ficheros regulares y, por lo tanto, hojas del árbol.

1.4.4 CONVENCIONES PARA NOMBRAR A UN DIRECTORIO O FICHERO

Todo fichero simple o dispositivo en Unix posee tanto un nombre de fichero como una ruta o camino.

1.4.4.1 Nombres de ficheros

Un nombre de fichero es una secuencia de uno a catorce caracteres distintos de "slash" (/). Todo fichero regular, directorio o dispositivo del sistema tiene un nombre. En un mismo directorio no puede haber dos ficheros con el mismo nombre y, en directorios diferentes, pueden haber dos ficheros con igual nombre.

Los caracteres alfanuméricos y el carácter punto constituyen caracteres válidos para la especificación del nombre de un fichero.

No se recomienda el uso de caracteres de control para especificar el nombre de un fichero o directorio. Cuando el carácter inicial del nombre de un fichero es el punto, se le considera "oculto" y no será incluido dicho fichero en el listado producido por el comando l$.

1.4.4.2 Ruta o camino de un fichero

Una ruta o un camino es una secuencia de nombres de directorio seguidos de un nombre simple de fichero. Cada nombre de un fichero está separado del anterior por un "slash". Existen dos tipos de direccionamiento de rutas:

a) Ruta con direccionamiento absoluto:

Si un nombre de una ruta comienza con un "slash" se interpretará como relativa a la raíz del sistema de ficheros.

@example:

disco2/acct/usr/dnr$s_t1/oficios

b) Ruta con direccionamiento relativo:

Si un nombre de una ruta comienza con un carácter diferente de "slash", se interpretará como relativa al directorio actual.

@example:
	dnr$s_t1/oficios
A continuación presentamos una serie de ejemplos de nombres de ficheros con rutas:

/ especifica el directorio raíz del sistema de ficheros.
/bin es el directorio que contiene los comandos más empleados en Unix.
/usr es el directorio que contienen los directorios personales de cada usuario definido por el sistema.
/dev es el directorio que contiene ficheros correspondientes a los dispositivos físicos del sistema (terminales, impresoras y discos).
/lib es el directorio que contiene los ficheros empleados por los compiladores de lenguajes de programación.

`fichero1` significa el nombre de un fichero regular dentro del directorio corriente.

Bajo el usuario Unix, cada usuario reside en un directorio llamado home directory. Todos los directorios tienen asociado un directorio "padre". Este es el directorio situado un nivel inmediatamente arriba del que se está analizando (directorio actual de trabajo).

Unix provee dos abreviaturas para los directorios padre y actual:

. El nombre del directorio actual. Así por ejemplo, ./texto hace referencia al fichero "texto" que existe en el directorio actual.
.. El nombre del directorio padre. Así por ejemplo, el nombre abreviado ../../../nombra al directorio que está dos niveles por encima del directorio actual.

### 1.4.4.3 Uso de metacaracteres

Unix provee una facilidad de identificación de patrones para especificar conjuntos de nombres de ficheros que cumplan con un formato particular. Existe tres metacaracteres que ayudan al usuario en la especificación de nombres de ficheros y son: el * (asterisco), la ? (interrogación), los [ ] (corchetes).

El metacaracter * (asterisco) reemplaza a cero o más caracteres dentro de la especificación del nombre de un fichero.

El metacaracter ? reemplaza a un solo carácter dentro de la especificación del nombre de un fichero.

El metacaracter [ ] reemplaza a cualquier carácter que se encuentre dentro de los corchetes.

**ejemplo:** si se digita en la línea de comandos:

```
$ ls cap* <RETURN>
```

se obtiene como resultado la lista de todos los ficheros cuyos nombres comienzan con la palabra 'cap'.

UNIX: Una puerta abierta a los sistemas abiertos
$ cat * <RETURN>
desplegará todos los ficheros del directorio actual en la pantalla.

$ ls cap[a-z]* <RETURN>
desplegará en pantalla todos los nombres de los ficheros que comiencen con 'cap' y además en la cuarta posición tengan una letra del abecedario en minúsculas, el resto de letras del nombre del fichero puede ser cualquiera.

$ ls ? <RETURN>
lista todos los ficheros cuyo nombre conste de un solo carácter.

$ ls -l cap?.1
listará los nombres de ficheros que comiencen con 'cap', tengan cualquier carácter en la cuarta posición y termine con .1, es decir, como resultado se podría tener nombres de ficheros como: cap1.1, cap2.1, cap3.1, cap4.1, capa.1, etc.

1.5 COMANDOS
Un comando es una orden ingresada y que es ejecutada por el ordenador. Los comandos se emplean para invocar programas ejecutables. Cuando se teclea un comando, Unix lee la línea de comandos digitada, busca un programa que tenga el nombre dado y lo ejecuta al encontrarlo.

Las líneas de comandos pueden también contener argumentos que especifiquen opciones o ficheros que el programa pueda necesitar.

1.5.1 SINTAXIS DE UN COMANDO
En el sistema operativo Unix la sintaxis general para un comandos será siempre:

    comando [opciones] [nombres de ficheros]

Por convención, los comandos tienen nombres en minúscula. Las opciones son indicativos que seleccionan varias modalidades de proceso disponibles para cada comando durante su ejecución. Las opciones consisten de un signo menos (-) y una letra de identificación. La especificación de las opciones, así como los nombres de los ficheros sobre los cuales va a trabajar el comando son opcionales; esto se indica en la sintaxis mediante los símbolos [ ].

    ejemplo:

    $date <RETURN>
    $time <RETURN>
    $who am i <RETURN>
1.5.2 REDIRECCIONAMIENTO

Todo comando en Unix procesa información de entrada y produce información de salida. Usualmente, los comandos toman la entrada de la terminal (teclado) o de algún fichero que se especifica para el comando, en cambio que la salida que produce un comando la presentan en la terminal (video) o en un dispositivo de salida (fichero, impresora).

En Unix, es posible que un fichero pueda reemplazar a la terminal, tanto para tomar los datos de entrada como para depositar los datos de salida.

El símbolo empleado para redireccionamiento de salida es > y significa "coloque la salida que produce la ejecución del comando en el siguiente fichero, en vez de en la pantalla de la terminal".

ejemplo:

⇒ $ cat fichero1 > resultado <RETURN>

significa que el despliegue del contenido del fichero1 va a un fichero de nombre resultado, en vez de salir a pantalla.

De forma similar, el símbolo de redireccionamiento de entrada < significa "tome la entrada para el comando del siguiente fichero, en vez desde el teclado de la terminal".

ejemplo:

⇒ $mail dnrs_t1 < resultado <RETURN>

quieres decir que el usuario envía un mensaje al login dnrs_t1, y que dicho mensaje se encuentra escrito en el fichero resultado.

1.5.3 RESUMEN DE COMANDOS

1.5.3.1 Manipulación de ficheros

1.5.3.1.1 Listado de ficheros

comando ls:

El comando ls determina los nombres de los ficheros y directorios residentes en un directorio dado y los lista alfabéticamente. Si no se especifica un directorio, lo listará el contenido del directorio actual de trabajo.

sintaxis:

ls [opciones] [nombre_directorio]

ejemplos:

⇒ para mirar los ficheros que tiene el directorio de trabajo actual se digita:

$ ls <RETURN>
⇒ para mirar el contenido del subdirectorio denominado cartas (ubicado en el directorio actual), se digita:

```
$ ls cartas <RETURN>
```

⇒ el comando ls con la opción -l (ls -l) imprime una lista "larga", la cual incluye la siguiente información para cada fichero o directorio: permisos, número de enlaces, dueño, grupo, tamaño en bytes, fecha de última modificación y nombre de fichero.

```
$ ls -l <RETURN>

$ ls -l cartas <RETURN>
```

### 1.5.3.1.2 Despliegue de ficheros

**comando cat:**

Este comando despliega el contenido de uno o varios ficheros, sin alterar los ficheros originales.

**sintaxis:**

```
cat nombre_fichero_1 ... nombre_fichero_n
```

Es frecuente emplear el comando cat en conjunción con el redireccionamiento de salida, con el propósito de combinar varios ficheros en uno solo.

**ejemplo:**

⇒ para colocar el contenido de dos ficheros, fichero1 y fichero2, en el fichero3.

```
$ cat fichero1 fichero2 > fichero3 <RETURN>
```

**comando more:**

El comando more despliega el contenido de uno o varios ficheros, una "página" a la vez (22 líneas). Es útil para revisar el fichero cuando no se desea introducir cambios en él.

**sintaxis:**

```
more nombre_fichero_1 ... nombre_fichero_n
```

Si el fichero contiene más de un desplegado (más de 22 líneas), en la última línea de la pantalla aparecerá impreso el porcentaje del fichero que queda por examinar. Para continuar desplegando más información del fichero emplee las siguientes teclas:

- **return:** adelanta una línea
- **d:** adelanta medio desplegado (11 líneas)
- **espacio:** adelanta un desplegado completo (22 líneas)
- **n espacio:** donde n es un número, adelanta n líneas
1.5.3.1.3 Copiado de ficheros

**comando cp (copy file):**

El comando `cp` crea una copia de un fichero. Como resultado de este comando, existen dos ficheros idénticos en contenido pero con nombres distintos.

**sintaxis:**

```plaintext
cp fichero_original fichero_copia
```

1.5.3.1.4 Renombre de ficheros

**comando mv (move):**

El comando `mv` cambia el nombre de un fichero. El contenido del fichero no es alterado. Este comando sirve también para efectuar el movimiento de un fichero a otro directorio.

**sintaxis:**

```plaintext
mv fichero_original fichero_nuevo
```

**ejemplo:**

```plaintext
⇒ este ejemplo cambia de nombre el fichero final por final. respaldo.
$ mv final final.respaldo <RETURN>
⇒ este ejemplo mueve el fichero final. respaldo del directorio actual al subdirectorio de nombre dnrs_t8
$ mv final.respaldo /disco2/usr/acct/dnrs_t8 <RETURN>
```

1.5.3.1.5 Borrado de ficheros

**comando rm (remove):**

El comando `rm` remueve (borra) los ficheros que se le indiquen. Una vez borrado un fichero, en Unix (Xenix) no hay manera de recuperar la información que ese fichero contenía.

**sintaxis:**

```plaintext
rm nombre_fichero_1 ... nombre_fichero_n
```

**ejemplo:**

```plaintext
⇒ este ejemplo borra los ficheros denominados fichero1, fichero2.
$ rm fichero1 fichero2
```

1.5.3.1.6 Cambio de permisos en los ficheros

**comando chmod (change mode):**
El comando chmod cambia los permisos de lectura, escritura, ejecución y búsqueda de un fichero o directorio. Este fichero es útil si se ha creado un fichero con un modo y se desea dar (o suprimir) a otros el permiso de leer, escribir o ejecutar dicho fichero.

sintaxis:
```
chmod instrucción nombre_fichero
```

El segmento instrucción del comando indica qué permisos se quieren cambiar para qué clases de usuarios. Hay tres clases de usuarios, descritos a continuación:

- **u**: User, dueño del fichero
- **g**: Group, grupo al que pertenece el dueño del fichero
- **o**: Other, todos los demás usuarios del sistema
- **a**: All, todas las clases de usuarios

Y además existen tres clases de permisos:

- **r**: Read, lectura. Permite examinar un fichero pero no cambiar ni borrar el fichero
- **w**: Write, escritura. Permite cambiar y borrar el fichero
- **x**: Execute, ejecución. Permite ejecutar el fichero como un comando

Ejemplo:

- el siguiente comando confiere permisos de escritura y ejecución, únicamente a los miembros del grupo, sobre el fichero denominado fichero1:
  ```
  $chmod g+wr fichero1 <RETURN>
  ```

- suprimir los permisos de escritura y ejecución tanto al usuarios (dueño) como al grupo del fichero denominado fichero1:
  ```
  $chmod ug-wx fichero1 <RETURN>
  ```

### 1.5.3.2 Manipulación de directorios

#### 1.5.3.2.1 Creación de directorios

**comando mkdir (make directory):**

Este comando sirve para crear directorios o subdirectorios.

sintaxis:
```
mkdir nombre_directorio1 ... nombre_directorio-n
```

Ejemplo:
1.5.3.2.2 **Borrado de directorios**

**comando rmdir (remove directory):**

Este comando sirve para borrar directorios o subdirectorios previamente creados por el usuario. Para poder borrar un directorio, es imprescindible que este esté vacío, es decir, que no contenga ficheros o subdirectorios.

**sintaxis:**

```
rmdir nombre_directorio1 ... nombre_directorio-n
```

**ejemplo:**

⇒ **borrar el subdirectorio de nombre documentos como hijo del subdirectorio dnrs_t1**

```
$ cd /disco2/usr/acct/dnrs_t1 <RETURN>
$ mkdir documentos <RETURN>
```

1.5.3.2.3 **Cambio de directorio**

**comando cd (change directory):**

Este comando sirve para cambiarse de un directorio a otro, es decir, permite navegar a través de la estructura de árbol del sistema de ficheros del sistema operativo UNIX.

**sintaxis:**

```
cd nombre_del_directorio_a_posicionarse
```

**ejemplo:**

⇒ **cambiarse al subdirectorio dnrs_t3 con direccionamiento absoluto:**

```
$ cd /disco2/usr/acct/dnrs_t3 <RETURN>
```

1.5.3.2.4 **Impresión del directorio actual**

**comando pwd (print work directory):**

Este comando sirve para indicar al usuario en qué directorio de trabajo se encuentra posicionado actualmente.

**sintaxis:**

```
pwd
```

**ejemplo:**
⇒ imprimir en la pantalla el directorio en el cual se encuentra el usuario colocado actualmente
$ pwd <RETURN>

1.5.3.3 Comunicación con otros usuarios

Puesto que Unix soporta múltiples usuarios, es muy conveniente comunicarse con otros usuarios del sistema. Los distintos métodos de comunicación se describen a continuación.

1.5.3.3.1 Envío de correo electrónico

comando mail:

El comando mail es una facilidad estándar en el sistema que permite que los usuarios envíen y reciban correo. Con este comando es necesario tanto un proceso para envío de mensajes como para la recepción de los mismos.

ejemplo:

⇒ para enviar correo al usuario juan, teclee lo siguiente:
$ mail juan <RETURN>
habrá una sesión de trabajo el día
sábado, a partir de las 10:00 am
<ctrl d>

Como se indica en el ejemplo anterior, a continuación de indicar a quien se desea enviar correo (mail juan) se debe indicar el mensaje a enviar; ctrl d es la indicación de fin de mensaje.

⇒ para enviar el mismo mensaje a varios usuarios, por ejemplo a los usuarios juan, pedro y maría, digite lo siguiente:
$ mail juan pedro maria <RETURN>
habrá una sesión de trabajo el día
sábado, a partir de las 10:00 am
<ctrl d>

⇒ si el mensaje se encuentra escrito en un fichero, por ejemplo en el fichero denominado monago, no hay necesidad de digitarlo después de indicar a quien se desea mandar el mensaje:
$ mail juan pedro maria < mensaje <RETURN>

Para leer el correo que le llega a un usuario, se debe teclear: mail.

En la terminal aparecerá una cabecera para cada mensaje. Cuando se teclee <RETURN>, aparecerá el contenido del primer mensaje que tenga el usuario.
Los mensajes siguientes (si existen más) serán desplegados, uno a la vez, cada vez que se presione return.

1.5.3.2 Envío de mensajes a una terminal

Comando write:

Este comando sirve para enviar mensajes desde una terminal de un usuario a otra terminal de otro usuario, los dos usuarios tienen que estar en disposición de recibir mensajes (msg y).

Sintaxis:

write identificador_de_usuario

Al utilizar el comando write, el mensaje aparece inmediatamente en la terminal del otro usuario.

Ejemplo:

⇒ para enviar un mensaje al usuario juan, teclee lo siguiente:

$ write juan <RETURN>

habrá una sesión de trabajo el día

sábado, a partir de las 10:00 am

<ctrl-d>

Como se indica en el ejemplo anterior, a continuación de indicar a quién se desea enviar el mensaje (write juan) se debe indicar el mensaje a enviar; <ctrl d> es la indicación de fin de mensaje.

1.5.3.4 Otros comandos de UNIX

Comando who:

Este comando sirve para indicar cuáles usuarios se encuentran conectados al sistema en ese momento

Sintaxis:

who [[am i] [-u]]

Ejemplo:

⇒ preguntar al sistema por la auto-identificación del usuario.

$who am i <RETURN>

⇒ preguntar al sistema cuántos usuarios están conectados al sistema en ese momento.

$who -u <RETURN>

Comando at:
Este comando sirve para indicar al ordenador que execute una orden a una hora determinada de un día determinado.

**Sintaxis:**

```
at [mes] [día] [hora]
```

comandos de UNIX a ejecutarse

```
<ctrl-d>
```

**Ejemplo:**

⇒ el día 14 de abril de 1992 a las 15 ejecute los comandos: ls -al, who -u, who am i, mail.

```
$ at 4 14 1500 <RETURN>
ls -al
who -u
who am i
mail
<ctrl-d>
```

**Comando grep:**

Este comando permite encontrar todas las ocurrencias de un patrón o cadena de caracteres especificado dentro de un fichero.

**Sintaxis:**

```
grep 'patrón_a_buscar' nombre_del_fichero
```

**Ejemplo:**

⇒ el siguiente ejemplo encuentra y presenta en pantalla todas las ocurrencias de líneas que contengan la palabra "juan" en el fichero teléfonos.

```
$ grep 'juan' teléfonos <RETURN>
```

**Comando sort:**

Sort presenta ordenas las líneas de un fichero en orden ASCII ascendente (esto es, en orden alfabético), sin alterar el fichero original.

**Sintaxis:**

```
sort nombre_fichero
```

**Ejemplo:**

⇒ presentación en pantalla del contenido ordenado del fichero teléfonos.

```
$ sort teléfonos <RETURN>
```
⇒ ordenamiento de las líneas del fichero teléfonos y almacenamiento en el fichero ordenado.

$ sort teléfonos > ordenado <RETURN>

comando wc (word count):
El comando wc es un utilitario utilizado para contar líneas, palabras, y caracteres existentes en un fichero. Se asume que las palabras están separadas mediante blancos, signos de puntuación o por "new lines".
sintaxis:

   wc nombre_fichero

ejemplo:
⇒ cuenta las líneas, palabras y caracteres del fichero denominado fichero1. Una salida típica que describe lo anterior sería:

   $ wc fichero1 <RETURN>
   44 500 48000 fichero1

comando cal (calendar):
El comando cal imprime el calendario del año especificado, o para cualquier mes del año especificado.
sintaxis:

   cal mes año

ejemplo:
⇒ para imprimir el calendario del año 1965:

   $ cal 1965 <RETURN>
⇒ para imprimir el calendarios del mes de octubre del año 1965:

   $ cal 10 1965 <RETURN>

1.6 EDITOR DE TEXTOS VISUAL (VI)
El editor de texto vi permite la creación y edición de texto. Dentro de vi existen dos modos de operación:

   v modo comando
   v modo de edición

Para ingresar a modo comando se utiliza la tecla <esc> y el comando a utilizar. Para ingresar al modo de edición se utiliza <<esc> i> o <<esc> a>

Sintaxis:

   vi nombre_de_fichero
1.6.1 COMANDOS MAS UTILIZADOS DENTRO DEL EDITOR VI

- Movimiento del cursor:
  - `<esc> <h>` una posición a la izquierda
  - `<esc> <j>` una línea abajo
  - `<esc> <k>` una línea arriba
  - `<esc> <l>` una posición a la derecha

- Ir a la línea n-ésima:
  - `<esc> <1> <G>` ir a la línea 1
  - `<esc> n <G>` ir a la línea número n del texto, siendo n un entero positivo

- Numerar líneas en el texto:
  - `<esc> <.: set nu` Permite numerar la líneas en el texto pero no escri- 
  - `- be la numeración dentro del texto.

- Eliminar la numeración visual del texto
  - `<esc> <.: set nonu`

- Avanzar hacia adelante media pantalla
  - `<ctrl-u>`

- Retroceder media pantalla
  - `<ctrl-d>`

- Avanzar hacia adelante una pantalla
  - `<ctrl-f>`

- Retroceder una pantalla
  - `<ctrl-b>`

- Copiar una línea de texto: se debe colocar el cursor en la línea que 
  desea copiar y presionar:
  - `<esc> <y> <y>`

  A continuación se coloca el cursor en la línea de texto donde se desea 
  poner la copia del texto y presionar:
  - `<p>` (para insertar antes de la línea de referencia),
  - `<P>` (para insertar después de la línea de referencia).

- Borrar una línea en el texto: se debe colocar el cursor en la línea que 
  desea borrar y presionar:
  - `<esc> <d> <d>`

UNIX: Una puerta abierta a los sistemas abiertos
v Copiar un bloque de texto: se debe colocar el cursor en la línea inicial del bloque a copiarse y presionar:

<esc> n <y> <y>

donde n es el número de líneas a copiar. A continuación se debe colocar el cursor en el lugar donde se quiera copiar el bloque (línea de referencia), y presionar:

<p> (para insertar antes de la línea de referencia),

<P> (para insertar después de la línea de referencia).

v Borrar un bloque de texto: se debe colocar el cursor en la línea a partir de la cual se eliminarán un número determinado de líneas, y presionar:

<esc> n <d> <d>, siendo n el número de líneas a ser borradas.

v Borrar un carácter dentro del texto: se debe posicionar el cursor sobre el carácter que desea eliminar y presionar:

<esc> <x>

v Insertar un carácter: se debe colocar el cursor un carácter antes del lugar en el que se desea insertar y presionar:

<esc> <i> c donde c es el carácter a insertar.

v Borrar una palabra: se debe posicionar el cursor en la palabra que se desea borrar, y presionar:

<esc> <d> <w> borra la palabra a la derecha del cursor

<esc> <d> <b> borra la palabra a la izquierda del cursor

v Mover un bloque de texto: se debe colocar el cursor al inicio del bloque de texto a ser movido, y a continuación presionar:

<esc> n <d> <d> donde n es el número de líneas del bloque que se desea mover

A continuación se debe colocar el cursor en la línea de referencia de texto sobre, o bajo, la cual se quiere que quede el bloque de texto a mover. Para ello se presiona

<p> bajo la línea de referencia

<P> sobre la línea de referencia

v Búsqueda de texto

<esc> <,> </> patrón Búsqueda hacia adelante

<esc> <,> </> ?patrón Búsqueda hacia adelante

donde patrón es el conjunto de caracteres a ser buscados.

Si desea seguir buscando el mismo patrón o cadena de caracteres a lo largo de todo el texto se utiliza el comando: <n> tanto para la búsqueda hacia adelante como para atrás del texto.
Sustitución de texto

<esc> <c> <c>

significa sustituir una línea entera por el texto digitado a continuación a través de teclado. Se termina la sustitución presionando <esc>

<esc> <c> <w>

significa sustituir una palabra por el texto ingresado a continuación a través del teclado. Se termina la sustitución con <esc>

<esc> <r> c

significa sustituir por c el carácter que se digita donde está el cursor.

Búsqueda y reemplazo de texto

sirve para cambiar todas las ocurrencias de un patrón1 por un patrón2 en el texto. El comando es el siguiente:

<esc> <:> 1,${patrón1}/${patrón2}/g

este comando quiere decir "Desde el principio de la primera línea (1) hasta el final del fichero ($) encuentre el patrón1 y reemplace por el patrón2 (s/patrón1/patrón2/) todas las veces que ocurra en cada línea del texto (g)"

Volver a estado anterior del texto (UNDO)

<esc> <u>
El shell es un lenguaje de programación completo con variables y control de flujo, además de ser un lenguaje intérprete de comandos. El shell no es un caso típico dentro de los intérpretes de comandos: aunque permite al usuario ejecutar comandos en la forma usual, por ser un lenguaje de programación, puede hacer mucho más. Algunas de las tareas que puede ejecutar son redireccionar E/S, iterar y expandir nombres de ficheros, y algo más importante: la aplicación de estos recursos es uniforme en todos los programas.

Otras características como los ficheros de shell y las interconexiones, son en realidad proporcionados por el núcleo, pero el shell proporciona una sintaxis natural para crearlos. Estos van más allá de la comodidad; sirven en realidad para incrementar las capacidades del sistema. La comunicación con el núcleo es bidireccional, cuando se ingresa un requerimiento al sistema, shell lo traduce a éste en un lenguaje que la máquina entiende. Si es válido se lleva a cabo una instrucción o set de instrucciones.

A un programa shell que consta de un conjunto de comandos se lo llama SHELL SCRIPTS.

### 2.1 COMANDOS BACKGROUND

Para ejecutar un comando el shell normalmente crea un nuevo proceso y espera a que este termine. Un comando puede ser corrido sin esperar a que éste termine.
Por ejemplo:

$ cc pgm.c &

llama al compilador c para compilar el fichero pgm.c, el caracter & es un operador que indica al shell no esperar para que el comando termine. Para guardar una pista de tal proceso el shell reporta un número de proceso siguiente a la creación.

Nosotros podemos utilizar un comando adicional nohup para ejecutar cualquier comando inmune a cortes o salidas del sistema. Todos los procesos que son ejecutados en background corren eficientemente conjuntamente con nohup.

Por ejemplo:

$ nohup cc pgm.c &

2.2 REDIRECCIÓN DE ENTRADA/SALIDA

La mayoría de comandos producen salida al dispositivo de salida standard, que es normalmente el terminal. Esta salida puede ser enviada a un fichero escribiendo lo siguiente:

ls -l > fichero

La notación > fichero es interpretada por el shell y no es pasada como un argumento a Is. Si el fichero no existe lo crea caso contrario lo reemplaza. Además de ello la salida puede ser añadida a un fichero usando la siguiente notación:

ls -l >> fichero

En este caso fichero también es creado si no existe, pero si existe la salida obtenida de ls -l es concatenada o añadida.

La entrada standard puede ser tomada también de un fichero en lugar de teclado, así por ejemplo:

wc < fichero

2.3 PIPES (CONDUCTOS) Y FILTROS

La salida de un comando puede ser concatenado a la entrada de otro comando escribiendo el operador pipe |, así por ejemplo:

ls -l | wc

los dos comandos conectados de esta manera constituyen un pipeline y tendría el mismo efecto que el siguiente comando:

ls -l > fichero
wc < fichero

Un filtro es un comando que lee la entrada standard, transforma en alguna manera la información e imprime el resultado.
2.4 EL LENGUAJE SHELL

Existen varios tipos de shell que son el Bourne Shell, c-Shell, korn shell, etc. Las tareas que se pueden hacer con shell son:

- Usar un solo comando para iniciar la ejecución de una serie de ellos
- Ejecutar rápidamente procesos repetitivos, en lugar de repetir un set
- Crear utilerías propias
- Hacer tareas más complicadas ocultas al usuario

2.4.1 CREACION DE UN SHELL

Usar un editor de texto para escribir los comandos necesarios dentro de él.

- Asegurarse que el fichero tiene permiso de ejecución
- Ejecutar sólo escribiendo el nombre del fichero

En general se puede usar el shell de UNIX como un intérprete de comandos (por ejemplo redirecciones I/O, pipes, filtros, etc), además de ello éste puede ser usado como un lenguaje de programación para ejecutar tareas complejas.

Por ello, el shell tiene elementos y los fundamentos necesarios para ser un lenguaje de programación. Usando estas características de shell se puede controlar el flujo de datos.

Un fichero puede ser creado cuyo contenido sea un conjunto de comandos UNIX que ejecutarán una tarea del mismo modo como si fueran ejecutados independientemente. Además de contener comandos UNIX es posible que este pueda contener estructuras de control, variables, argumentos, etc.

El Bourne shell fue desarrollado en los Laboratorios Bell, además de ella existe la versión de C-Shell desarrollado en la Universidad de Berkeley, el Korn Shell, etc.

2.4.2 EJECUCION DE FICHEROS SHELL

Los procedimientos shell pueden ser ejecutados de dos maneras, la primera de ellas es usar el comando sh seguido por el nombre del fichero.

```
sh nombre_shell
```

Es posible ejecutar un shell en modo de depuración y se lo hace mediante el comando siguiente:

```
sh -v nombre_shell
```

o alternativamente se puede signar dentro del shell una asignación a una bandera para que active el modo de depuración y se lo hace del siguiente modo:

```
set -v
```
La segunda alternativa de ejecutar un procedimiento shell es hacerlo ejecutable, esto es, poner en el modo adecuado de ejecución mediante el comando chmod, así por ejemplo:

```
chmod 700 nombre_shell
```

el ejecutarlo sería tan solo una llamada al nombre del fichero.

### 2.4.3 VARIABLES SHELL

Una variable es un nombre representando un valor tipo cadena. Ciertas variables son usualmente referidas como parámetros. Otras variables son nombres a las cuales el usuario o el shell asignan valores.

Shell tiene cuatro métodos de crear variables:

- Mediante asignaciones (=)
- Mediante parámetros posicionales
- Mediante set
- Mediante read

#### 2.4.3.1 Mediante asignaciones

Los nombres de variables deben empezar con una letra y puede consistir de letras, dígitos y underscores (_). Las variables pueden ser dadas valores mediante asignación usando el signo `=.`

**Por ejemplo**

```
usuario=juan numero=123 accno=mrr000
```

Nota que no existen espacios entre los dos miembros de la asignación. Una variable puede ser inicializada a un valor nulo usando lo siguiente:

```
var_nula=
```

Una variable puede ser substituida o referida precediendo su nombre con el signo `$`. En el ejemplo siguiente el valor de la variable usuario será desplegado mediante el comando echo.

```
usuario=juan
echo $usuario
```

#### 2.4.3.2 Mediante parámetros posicionales

Cuando un procedimiento shell es invocado el shell implícitamente crea parámetros posicionales, el argumento en posición cero es el procedimiento shell mismo(o fichero shell) y es referenciado mediante `$0`, el primer argumento es referenciado `$1`, el segundo `$2`, etc hasta un máximo de 9.

**Por ejemplo**

```
prueba juan
```
2.4.3.3 Mediante set

También un usuario explícitamente puede forzar el asignar valores en los parámetros posicionales, usando el comando set dentro del procedimiento shell.

Por ejemplo:
```
set abc def ghi
```

asigna el string abc al parámetro 1, def al parámetro 2 y ghi al parámetro 3.

2.4.3.4 Mediante read

Finalmente las variables puede ser dadas valores indirectamente usando el comando read. Este comando toma una línea de la salida standard (usualmente el terminal) y asignan consecutivamente palabras a las variables nombradas.

Por ejemplo:
```
read var1 var2 var3
```

si la entrada uno dos tres el efecto será el siguiente: var1 tendrá el valor uno, var2 el valor dos y var3 el valor tres.

2.4.4 DESPLIEGUE DE SALIDA

La salida de los procedimientos shell pueden ser redireccionados a terminal, a impresora o a otros ficheros usando la salida standard, comandos shell o caracteres especiales de redirección de I/O.

Si el usuario desea desplegar string literales, diagnósticos, etc el comando echo puede ser usado. Echo escribe sus argumentos separados por blancos y terminados por nueva línea. Este también usa algunas convenciones del lenguaje C como los caracteres de escape, una lista parcial tenemos a continuación

```
v b backspace
v c despliega línea sin saltar a la siguiente
v f form-feed
v n nueva línea
```

El comando echo es útil para desplegar el contenido de una variable shell dado un valor en una sentencia de asignación o para desplegar parámetros posicionales pasados al procedimiento shell desde la línea de comandos.
2.4.5 **COMANDOS USADOS EN UN SHELL**

2.4.5.1 Referencia a comandos de Unix

Cualquier comando puede ser referenciado usando el caracter ` de esta manera se captura la salida del comando en una variable.

*Por ejemplo:*

\[
\text{fecha} = \text{`date`}
\]

\[
\text{echo $fecha}
\]

asigna el valor de la fecha actual a la variable fecha y la despliega.

En general cualquier conjunto de comandos pueden ser encerrados usando este caracter especial.

*Por ejemplo*

\[
\text{usuarios} = \text{`who | wc -l`}
\]

2.4.5.2 Comando test

La Estructura test aunque no es parte del shell es aceptado para ser usado en los programas shell. A continuación se indica algunos de los argumentos usados con el comando test

- `test -s fichero` true si el fichero existe y no está vacío
- `test -f fichero` true si fichero existe y es un fichero regular
- `test -r fichero` true si fichero es leíble
- `test -w fichero` true si fichero se puede escribir
- `test -d fichero` true si fichero es directorio
- `test s1 = s2` true si string1 igual a string2
- `test n1 -eq n2` true entero n1 igual a entero n2

Esta estructura generalmente es usado con las sentencias condicionales. Se puede reemplazar la palabra test por los caracteres corchetes ([...]).

*Por ejemplo:*

\[
\text{test -f $1 es equivalente a [ -f $1 ]}
\]

2.4.5.3 Estructuras de control

Shell suministra un conjunto de estructuras de control que permite la manipulación lógica de operaciones en un programa shell. Se suministra estructuras de control condicionales repetitivas y secuenciales, elementos básicos para elaborar programas suficientemente poderosos.
### 2.4.5.3.1 Sentencia CASE

Una estructura de múltiples saltos es suministrado por la sentencia case, el formato de esta sentencia se muestra a continuación:

```bash
case $variable in
  patron1)
    comando1
    comando2
    ......
    comandoN;;
  patron2)
    comandoL
    comandoM
    ......
    comando K;;
  .......
  *)
    comandoA
    .......;
esac
```

Shell trata de instanciar la variable con algún patrón. esto es, si variable se instancia con el patrón1 se ejecutan todos los comandos correspondientes a esa opción, de igual manera si se instancia con algún otro patrón, si no se instancia con ninguna opción no sucede nada. Note que sólo un patrón es instanciado. Un patrón especial es el asterisco * el cual ejecuta la serie de comandos si la variable analizada no se instanció con ningún patrón.

En los patrones indicados es posible usar los caracteres de especial significado en ambiente shell (metacaracteres) como son *, ?.

### 2.4.5.3.2 Sentencia FOR

La estructura for es para ejecutar algún conjunto de operaciones por cada fichero en un conjunto de ficheros o ejecutar un comando una vez por cada elemento de los varios argumentos. El formato de esta sentencia es:

```bash
for variable in lista_palabras
do
  lista_comandos
done
```

donde lista_palabras es un conjunto de strings separados por blancos.

Los comandos en la lista de comandos son ejecutados una vez por cada palabra en la lista_palabras.

La parte de "in lista_palabras" puede ser omitida. Esto causará que el actual conjunto de parámetros posicionales a ser usados en lugar de la lista_palabras. Esto es conveniente para escritura de lazos que ejecutan el mismo conjunto de comandos para un número de variables.
2.4.5.3.3 Sentencia IF

Shell suministra estructuras condicionales mediante la sentencia IF. La forma más simple tiene la siguiente forma:

```
if lista_comandos then
    lista_comandos
fi
```

Los comandos siguientes al if son ejecutados si el último comando en la lista retorna true en su código de salida. El string fi indica el final de la estructura if.

La mayoría de estructuras son hechas mediante el comando test para evaluar la relación condicional.

La cláusula else puede ser usada en el caso de que el comando verificado tuvo un código de retorno falso. Posee la siguiente estructura:

```
if lista_comandos then
    lista_comandos
else
    lista_comandos
fi
```

2.4.5.3.4 Sentencia WHILE

La estructura de la sentencia WHILE tiene la siguiente forma:

```
while lista_comandos
    do
        lista_comandos
    done
```

El valor evaluado por el comando while es el status del último comando siguiente al while. Cada vez que se pase por la estructura while se ejecutarán los comandos indicados, estos se ejecutarán mientras el código de retorno del while sea true.

2.4.5.3.5 Sentencia UNTIL

Si es usado until la condición es al revés del lazo while. Este es útil en casos en donde la lista de comandos deben esperar hasta que suceda un evento especial.

2.4.5.4 Comandos BREAK y CONTINUE

El comando break puede ser usado para terminar la ejecución de un lazo while o un lazo for. El comando continue causa la ejecución de la siguiente
iteración del lazo. Los dos comandos son efectivos sólo cuando ellos
aparecen entre las palabras clave do y done.

El comando break termina la ejecución de los lazos más internos, causando la
ejecución después del siguiente done.

El comando continue causa la ejecución de la siguiente iteración dentro del
lazo.

### 2.4.5.5 Comando EXPR

El comando expr puede ser también usado en los shell scripts para realizar
operaciones aritméticas sobre valores, así tenemos los siguientes operadores:

- `v +` suma
- `v -` resta
- `v *` multiplicación
- `v /` división entera
- `v %` resto

### 2.4.5.6 Comando ECHO

El comando echo es usado para desplegar mensajes o variables a terminal,
referenciándolas de la manera adecuada a las variables esto mediante el
signo `$variable`.

Además de ellos puede ser usado conjuntamente con ciertos caracteres C
para controlar diversas situaciones como son:

- `v \c` detiene el despliegue del monitor hasta donde encontró estos
caracteres.
- `v \b` Al ser usado obtenemos un retroceso de carácter
- `v \n` Es usado para salto de línea
- `v \f` Es usado para la llamada form-feed,
- `v \t` Es el tab
- `v \v` Es el tab vertical

### 2.4.5.7 Comando SHIFT

Se ha indicado que un pseudo limitante en cuanto a parámetros es el número
de parámetros posicionales a los cuales podemos referirnos, pero no llega a
ser cierto con el uso del comando shift que es un desplazador de los
parámetros, por lo cual se referencia cualquier número de parámetros,
desplazándoles el siguiente parámetro, lo tendrá disponible y podrá ser
referenciado como `$1`. El desplazamiento tiene efecto a partir del primer
parámetro posicional.
2.4.5.8 Comando TRAP

El uso de interrupciones es permitido también en shell y se lo hace mediante el comando trap que tiene la siguiente sintaxis:

\[\text{trap arg señal}\]

Donde arg es un string a ser interpretado como una lista de comandos y señal consiste de una o más números de señal. A continuación tenemos la lista de señales usadas en UNIX:

- 01 hangup
- 02 interrupt
- 03 quit
- 04 illegal instruction
- 05 trace trap
- 06 IOT instruction
- 07 EMT instruction
- 08 floating point exception
- 09 kill
- 10 bus error
- 11 segmentation violation
- 12 bad argument to system call
- 13 write on a pipe with no one to read it
- 14 alarm clock
- 15 software termination signal
- 16 user defined signal 1
- 17 user defined signal 2
- 18 death of a child
- 19 power fail

2.4.5.9 Comando EXIT

Con este comando salimos o terminamos el shell su sintaxis es la siguiente:

\[\text{exit [n]}\]

donde n es una señal de usuario que puede ser asignada para ser capturada por cualquier proceso, el número de la señal va desde 1 hasta 255.
2.4.6 OPERADORES ESPECIALES

Como todo operador están asociados a operandos y en este caso los operandos son variables o valores, estos operadores son usados en las estructuras condicionales tipo if, while, until, junto con el comando test, y así tenemos los siguientes:

- **en** no es igual a
- **gt** mayor que
- **ge** mayor igual
- **lt** menor que
- **le** menor igual
- **eq** es igual a
- **!=** no es igual (usado en strings)
- **!** operador unario de la negación
- **-** operador AND
- **-o** operador OR

2.4.7 VARIABLES ESPECIALES EN SHELL

Existen ciertos variables especiales de las cuales podemos obtener información muy importante y son las siguientes:

- **#$** Al referenciar esta variable se tiene por resultado el número de parámetros posicionales externos.
- **$?** Al referenciarlo obtenemos el código de status o retorno del último comando ejecutado.
- **$$** Al referenciarlo me retorna el número de proceso actual, el cual es un número de 5 dígitos, usualmente es referenciado para crear ficheros con la terminación de este número para que el fichero creado sea único.
- **$!** Contiene la identificación o número de proceso del último de ellos ejecutado en background.

2.4.8 VARIABLES SHELL

- **PS1** (CON MAYUSCULA) Es el prompt principal
  
  Ej: `PS1="STRING"`

- **PS2** (CON MAYUSCULA) Es el prompt secundario
  
  Ej: `PS2="string"`

- **HOME** El directorio hogar
  
  Ej: `HOME=/usr/acct/ejemplo`

UNIX: Una puerta abierta a los sistemas abiertos
v CDPATH Los caminos de búsqueda alternativos de subdirectorios, sin necesidad de cambiarse repetitivamente.

Ej: `CDPATH=/usr/acct/EXAMPLE/prueba`

v PATH Es usado para indicar el directorio de búsqueda de comandos

Ej: `PATH=/bin:/usr/bin`

En este ejemplo están dos paths, cuando existen varios paths se los separa mediante los dos puntos `:`

### 2.5 GRAMATICA DE SHELL

**item:** word
input-output
name = value

**simple-command:** item
  simple-command item

**command:** simple-command
  (command-list)
  {command-list}
  for name do command-list done
  for name in word ... do command-list done
  while command-list do command-list done
  until command-list do command-list done
  case word in case-part ... esac
  if command-list then command-list else-part fi

**pipeline:** command
  pipeline | command

**andor:** pipeline
  andor && pipeline
  andor || pipeline

**command-list:** andor
  command-list;
  command-list &
  command-list ; andor
  command-list & andor

**input-output:** > file
  < file
  >> word
  << word

**file:** word

UNIX: Una puerta abierta a los sistemas abiertos
& digit
& -

case-part: pattern) command-list;;

pattern: word
    pattern | word

else-part: elif command-list then command-list else-part
    else command-list
    empty

empty:

word: una secuencia de caracteres no blancos

name: Una secuencia de letras,dígitos o underscores empezando con una letra.

digit: 0 1 2 3 4 5 6 7 8 9

2.6 META CARACTERES Y PALABRAS RESERVADAS.

a) SINTÁCTICO
    | símbolo PIPE
    && símbolo andf
    || símbolo orf
    ; separador de comandos
    ;; delimitador case
    & comandos en background
    () agrupación de comandos
    < redirección de entrada
    > creación de salida
    >> añadir en salida

b) PATRONES
    * equiparar cualquier carácter, incluido el vacío
    ? equiparar un solo carácter
    [...] equiparar cualquier carácter indicado

c) SUBSTITUCION
    ${[...]} substituto de variable shell

UNIX: Una puerta abierta a los sistemas abiertos
... substituto de comando de salida

d) PALABRAS RESERVADAS
   if then else elif fi
   case in esac
   for while until do done
   {}

2.7 PROGRAMAS EJEMPLO

⇒ Asignación de variables y despliega variables
   city=Dayton
   location=$city-USA
   echo $location

⇒ Uso de comandos UNIX en shell y parámetros externos o posicionales ($1 es el parámetro positional externo primero)
   echo "Terminal" $1
   who|grep $1

⇒ Uso de parámetros posicionales y lectura de parámetros mediante read
   echo $3 $2 $1
   read first init last

⇒ Uso de comandos UNIX dentro de procedimientos shell
   autos=`cat tipos.car`
   echo $autos
   en donde tipos.car es un fichero que contiene varias marcas de coches, por ejemplo: mercedes, ford, etc.

⇒ Uso de if para desplegar un fichero lo valida si existe o no.
   if test $1 = "" then
     echo "Deber dar un nombre"
   else
     cat $1
   fi

⇒ Uso de case y parámetros posicionales al primero que puede se list o sort.
   case $1 in
     list) cat $2;;
sort) sort $2;;

*) echo "Error en el comando"
esac

⇒ Uso de case con metacaracteres, validando el primer parámetro.
case $1 in
  *.c) cc $1;;
  *.pas) echo " fuente no es C";;
  *) echo "parámetro erróneo"
esac

⇒ Uso de for con parámetros posicionales externos
for i do
grep $i teléfonos
done
en donde teléfonos es un ficheros con nombres y teléfonos en donde se busca.

⇒ Ejemplo de búsquedas de patrones en ficheros mediante grep
for i do
  if grep $i teléfonos then
    echo " existe"
  else
    echo " nombre $i no encontró "
  fi
done

⇒ Despliegue de todos los ficheros usando estructuras while
while test -f * do
  cat *
done

⇒ Este programa espera hasta que aparezca un fichero que tiene por nombre a.out que lo creará algún proceso.
until test -f a.out do
  sleep 300
done
ls

UNIX: Una puerta abierta a los sistemas abiertos
Uso de break y continue, termina el programa cuando se ingresa quit.

```bash
while true do
    echo "Ingrese datos"
    read dato
    case $dato in
        quit) break;;
        *) continue;;
    esac
done
```

Programa para copiar pares de ficheros

```bash
while test $2 != "" do
    cp $1 $2
    shift; shift
done
if test $1 != "" then
    echo "Parámetros impares"
fi
```

Fichero para imprimir ficheros

```bash
if test $# != 0 > /dev/null then
    ls > ficheros
    for i do
        if test -f $i > /dev/null then
            cat $i > /dev/lp00
        else
            echo $i: "Nombre de ficheros erróneo"
        fi
    done
    rm ficheros
else
    echo " Se requiere al menos un fichero para imprimir"
fi
```

UNIX: Una puerta abierta a los sistemas abiertos
Capítulo 3
LENGUAJE DE PROGRAMACIÓN AWK

Awk es un lenguaje de programación para procesamiento de ficheros, diseñado para que muchas tareas comunes de manipulación de texto y recuperación de información sean fáciles de realizar.

- Genera reportes
- Equipara patrones
- Valida datos
- Filtra datos para transmisión

3.1 ESTRUCTURA DEL PROGRAMA
Un programa awk es una secuencia de sentencias distribuida de la siguiente forma:

```
patrón { acción }
patrón { acción }
```

... 

El programa awk se ejecuta sobre un conjunto de ficheros de entrada. La operación básica de awk es examinar un conjunto de líneas de entrada, en orden y una a la vez. En cada línea de entrada awk busca los patrones especificados en el programa awk, entonces, si son encontrados se ejecuta la acción correspondiente a ese patrón. De esa manera, todas las sentencias padrón-acción del programa awk son procesadas para una línea de entrada...
dada. Cuando todos los patrones han sido examinados, se toma la siguiente línea de entrada; y el programa awk se ejecuta otra vez desde el principio del mismo.

En el comando awk, se puede omitir el patrón o la acción, pero no los dos a la vez. Si no existe acción para un patrón, la línea que equipara es impresa. Si no existe patrón para la acción, la acción es ejecutada para cada línea de entrada. Las acciones son encerradas entre llaves ( {, } ) para distinguirlas de los patrones, y pueden estar constituidas por más de una sentencia separadas por punto y coma o nuevas líneas.

Por ejemplo, el siguiente programa en awk, imprime cada línea de entrada que tenga una “x” en ella.

```awk
/x/ {print}
```

Un programa awk tiene la siguiente estructura:

- Una sección `<BEGIN>`
- Una sección principal o de `<registro>`
- Una sección `<END>`

La sección `<BEGIN>` es ejecutada antes de que cualquier línea de entrada sea leída, y la sección `<END>` después de que todos los ficheros de datos sean procesados. La sección de `<registro>` es un manejador de datos. Esto es, la sección que se ejecuta separadamente sobre cada línea de entrada. Las palabras BEGIN y END, son reconocidas como patrones por awk.

3.2 CONVENCION LEXICA

Todos los programas awk están formados por unidades léxicas denominadas tokens. En awk hay ocho tipos de tokens:

- Constantes numéricas
- Constantes string
- Palabras claves (reservadas)
- Identificadores
- Operadores
- Tokens de registro y de campo
- Comentarios
- Separadores

3.2.2 IDENTIFICADORES
Los identificadores en awk sirven para denotar variables y arreglos. Un identificador es una secuencia de letras, dígitos, y carácter de subrayado. Letras mayúsculas y minúsculas son diferenciadas en awk.

3.2.3 COMENTARIOS

Un comentario es introducido por un # y terminado por una nueva línea. Por ejemplo:

    # esta parte es un comentario

Un comentario puede ser añadido al final de cualquier línea de un programa awk.

3.2.4 SEPARADORES Y LLAVES

Los tokens en awk son separados usualmente por secuencias no nulas de blancos, tabuladores, nuevas líneas; ó por otros símbolos de puntuación tales como comas y puntos y comas;

Las llaves {...} agrupan acciones, slashes /.../ agrupan patrones de expresiones regulares, y comillas "..." encierran strings.

Nota: Los demás tokens serán mencionados en lo posterior.

3.3 USANDO AWK

Existen dos maneras en las cuales se puede presentar un programa awk de sentencias patrón-acción para su procesamiento:

a) Si el programa es corto (una línea o dos), a menudo es más fácil hacer que el programa sea el primer argumento en la línea de comandos

    awk 'programa' ficheros

donde ficheros es una lista opcional de ficheros de entrada y programa es el programa en awk. Note que existen apóstrofes alrededor del programa, de tal manera que el shell acepte el string completo (programa) como el primer argumento de awk. Ejemplo:

    awk '/x/ { print }' ficheros

b) Alternativamente, si el programa awk es largo, es más conveniente colocar el programa en un fichero separado, e indicar a awk que tome la sentencias patrón-acción de ese fichero. Esto se hace usando la opción -f después del comando awk. Ejemplo:

    awk -f awkprog ficheros

En donde awkprog es el nombre del fichero que contiene el programa awk.

Si no se especifican ficheros de entrada, awk usa la entrada estándar. Si se desea que la entrada sea tomada tanto de ficheros determinados
como de la entrada estándar, después de especificar los nombres de los ficheros de entrada se debe escribir el carácter - (menos).

*Por ejemplo:*

```
awk 'programa' ficheros -
awk -f awkprog ficheros -
```

### 3.4 PATRONES

Los patrones controlan la ejecución de las acciones: cuando un patrón equipara, su acción asociada es ejecutada.

Existen seis clases de patrones, para los cuales hay condiciones específicas de equiparamiento.

#### 3.4.1 BEGIN y END

Los patrones BEGIN y END, no equiparan ninguna línea de entrada. Las sentencias en la acción de BEGIN son ejecutadas antes de que la primera línea del primer fichero haya sido leída, y las acciones del padrón END son ejecutadas después de que la última línea del último fichero ha sido procesado. De esta manera, BEGIN y END proveen una manera de efectuar inicializaciones y conclusiones, respectivamente.

Los patrones BEGIN y END no se combinan con otros patrones; si existe más de un BEGIN, las acciones asociadas son ejecutadas en el orden en el cual aparezcan en el programa, y similarmente para múltiples END.

Aunque no es obligatorio, se puede poner BEGIN primero y END después.

Un uso común de una acción BEGIN es cambiar la manera por omisión en que las líneas de entrada (registros) son divididas en campos (el indicador de separador de campo es controlado por la variable inter-construida FS).

#### 3.4.2 EXPRESIONES COMO PATRONES

Como la mayoría de lenguajes, awk es rico en expresiones para describir cálculos numéricos. Además, cuenta con expresiones para describir operaciones sobre strings. Una expresión puede ser usada como un operand de cualquier operador.

Si una expresión tiene un valor numérico pero un operador requiere un valor string, el valor numérico es automáticamente transformado en un string; similarmente, un string es convertido en un número cuando un operador demanda un valor numérico. Cualquier expresión puede ser usada como un patrón.

Si una expresión usada como un patrón tiene un valor diferente de cero o diferente de nulo en la línea actual, entonces el patrón equipara esa línea.

Los patrones expresiones típicos son aquellos que involucran comparaciones entre números o strings.
Una expresión de comparación contiene uno de los seis operadores relaciones, o uno de los dos operadores de equiparación de string.

3.4.3 PATRONES DE EQUIPARACION DE STRING

Un patrón de equiparación de strings examina si un string contiene un substring equiparado por una expresión regular. La expresión regular más simple es un string de letras y números, como Ecuador, que equipara a sí mismo.

Para cambiar de una expresión regular a un patrón para equiparación de strings, encerramos la expresión regular entreslashes (/), ejemplo:

/ /Ecador/ 

Este patrón equipara cuando la línea de entrada actual contenga el substring Ecuador, sea como Ecuador en sí mismo o como parte de un string mayor (ejemplos, Ecuador-Turs, Quito-Ecuador, Sud-America-Ecuador-Quito). Note que los blancos son significativos en las expresiones regulares, ejemplo: el padrón de equiparación de strings

/ / Ecuador / 

equipara sólo cuando Ecuador está rodeado por blancos.

El patrón anterior es uno de los tres tipos de patrones para equiparación de strings.

En general, el primer tipo de patrón para equiparación de strings es una expresión r encerrada porslashes :

/r/ 

Este patrón equipara una línea de entrada si la línea contiene un substring equiparado por r.

Los otros dos tipos de patrones para equiparación de string usan un operador de equiparación explícito :

expresión ~ /r/ 

expresión !~ /r/ 

El operador de equiparación ~ significa "es equiparado por", y !~ significa "no es equiparado por".

El primer patrón equipara cuando el valor string de la expresión contiene un substring equiparado por la expresión regular r; ejemplo :

$4 ~ /Ecuador/ 

equipara todas las líneas de entrada en las cuales el cuarto campo contenga Ecuador como substring.

El segundo patrón equipara si no existe tal substring. Ejemplo :

$4 !~ /Ecuador/
3.4.4 EXPRESIONES REGULARES

Una expresión regular es una notación para especificación y equiparación de strings. Como una expresión aritmética, una expresión regular es una expresión básica o una creada al aplicar operadores a expresiones componentes.

Los caracteres:
\^ $ . " [ ] ( ) * + ?

son denominados metacaracteres debido a que tienen un significado especial.

Una expresión regular consistente de un carácter simple (un no metacaracter) equipara a sí mismo. Así, una letra simple o un dígito es una expresión regular que equipara con sí mismo. Ejemplos: A equipara con A, 1 con 1, etc.

Para preservar el significado literal (los significados originales) de los metacaracteres en una expresión regular, se los debe preceder por un backslash (\). Así, la expresión regular \$ equipara con el carácter $. Si un carácter es precedido por un único \, decimos que ese carácter está escapado.

En una expresión regular, el carácter ^ no escapado (es decir, el metacaracter) equipara con el comienzo de un string. El carácter $ no escapado equipara con fin de string; y, un punto (.) no escapado equipara con un carácter cualquiera.

Ejemplos:
\Rightarrow ^C \quad equipara a C al comienzo de un string.
\Rightarrow C$ \quad equipara con C al final de un string.
\Rightarrow ^C$ \quad equipara el string consistente de un solo carácter C.
\Rightarrow ^.$ \quad equipara cualquier string conteniendo exactamente un carácter.
\Rightarrow ^*.$ \quad equipara cualquier string que contiene exactamente tres caracteres.
\Rightarrow ...$ \quad equipara tres caracteres consecutivos cualesquiera.
\Rightarrow l.$ \quad equipara un punto (.) al final de un string.

Una expresión regular consistente de un grupo de caracteres encerrados entre [ ] (corchetes) es denominada una clase de caracteres.

Una clase de caracteres equipara con uno de los caracteres encerrados entre [ ]. Por ejemplo, [A,E,I,O,U] equipara cualquiera de los caracteres A,E,I,O,U.

En una clase de caracteres, se puede abreviar un rango de caracteres usando el signo de separación (-). El carácter situado a la izquierda del signo
de separación define el principio del rango, y el carácter situado inmediatamente a la derecha del signo define el final del rango.

Así, [0-9] equipara cualquier dígito, y [a-zA-Z][0-9] equipara a letra o dígito.

Sin carácter izquierdo y derecho, en una clase de caracteres, el carácter de separación (-) denota a sí mismo. Ejemplo: [+-] y [-+] equipara a + o a -; la clase de caracteres [A-Za-z-] equipara a una letra o al carácter de separación (-).

Una clase de caracteres complementada es una en la cual el primer carácter después de [ es un ^ (luego de un corchete izquierdo un sombrerito). Tales clases equiparan cualquier carácter que no esté en el grupo de caracteres especificados después de ^ hasta ]. Ejemplos: [^0-9] equipara cualquier carácter excepto un dígito; [^a-zA-Z] equipara cualquier carácter excepto una letra mayúscula o minúscula.

**Ejemplos:**

\[ ^[ABC] \] equipara una A, B, ó C al comienzo de un string.

\[ ^[^ABC] \] equipara cualquier carácter al comienzo de un string, excepto una A, B ó C.

\[ ^[ABC] \] equipara cualquier carácter diferente de A, B, ó C.

\[ ^[^a-zA-Z] \] equipara cualquier string de un solo carácter, excepto una letra minúscula.

En una clase de caracteres (dentro de [ ]), todos los caracteres tienen su significado literal (a significa a, 1 significa 1, $ significa $, punto significa punto, etc.) excepto los caracteres escapados (precedidos de \), ^ al comienzo de la clase, y - entre dos caracteres. Así, [ ] equipara un punto y[^[]] equipara cualquier caracter al principio de un string, excepto el ^.

Los paréntesis ( ) son utilizados en expresiones regulares para especificar cómo sus componentes son agrupados.

Existen dos operadores binarios para expresiones regulares: alternativa y concatenación.

El operador de alternativa | es utilizado para especificar alternativas: si r1 y r2 son expresiones regulares, entonces r1 | r2 equipara cualquier string que sea equiparado por r1 o por r2.

No existe un operador explícito de concatenación. Si r1 y r2 son expresiones regulares, entonces (r1)(r2) (sin blancos entre (r1) y (r2)) equipara cualquier string de la forma xy donde r1 equipara x y r2 equipara y. Los paréntesis alrededor de r1 o r2 pueden ser omitidos, si las expresiones regulares contenidas no contienen operadores de alteración.

Los símbolos *, +, y ? son operadores unarios usados para especificar repeticiones en las expresiones regulares.
Si r es una expresión regular, entonces (r)*, equipara cualquier string consistente de cero o más substrings consecutivos equiparados por r. (r)+ equipara cualquier string consistente de uno o más substrings consecutivos equiparados por r. Y, (r)? equipara el string nulo o cualquier string equiparado por r.

Si r es una expresión básica, los paréntesis pueden ser omitidos.

**Ejemplos:**

⇒ B* equipara el string nulo, o B o BB o BBB, etc.
⇒ AB*C equipara AC, o ABC, o ABBC, o ABBBC, etc.
⇒ AB+C equipara ABC (por lo menos), o ABBC, o ABBBC, etc.
⇒ ABB*C equipara ABC (por lo menos), o ABBC, o ABBBBC, etc.
⇒ AB?C equipara AC, o ABC.
⇒ [A-Z]+ equipara cualquier string de uno o más caracteres de letras mayúsculas.
⇒ (AB)+C equipara ABC (por lo menos), o ABABC, o ABABABC, etc.

En las expresiones regulares, el operador de alternativa |, tiene la menor precedencia, luego el de concatenación, y finalmente los operadores de repetición *, + y ?. Como en las expresiones aritméticas los operadores de mayor precedencia se efectúan primero que los de menor precedencia.

A continuación se presentan algunos ejemplos útiles de patrones para equiparación de strings que contienen expresiones regulares con operadores unarios y binarios. Recuerde que un patrón para equiparación de strings rt/ equipara la línea de entrada actual si la línea contiene al menos un substring equiparado por r.

⇒ /[^0-9]+$/
    equipara cualquier línea de entrada que consiste únicamente de dígitos (por lo menos uno).
⇒ /[^0-9][0-9][0-9]$/
    exactamente tres dígitos en la línea.
⇒ /[^+\-][0-9]+.?[0-9]*$/
    un número decimal con un signo opcional y con una parte fracción también opcional.
⇒ /[^+\-][0-9]+[.][0-9]*$/
    idéntico al anterior.
⇒ /[^+\-][0-9]+[.][0-9]*[Ee][+\-][0-9]+$/
    un número de coma flotante con signo opcional y exponente opcional (si hay exponente, el signo del exponente es opcional).

UNIX: Una puerta abierta a los sistemas abiertos
una letra seguida de cero o más letras o dígitos.
\n\Rightarrow /^[A-Za-z][A-Za-z0-9]*$/
una letra, ó una letra seguida de un dígito.
\n\Rightarrow /^[A-Za-z][0-9]$/
también una letra, ó una letra seguida de un dígito.

Ya que + (mas) y . (punto) son metacaracteres, deben ir precedidos de backslashes ( `\` ) en el tercer ejemplo para equipara la ocurrencia literal (es decir para que punto equipare con punto y que mas equipare con mas).

Dentro de expresiones regulares y strings, awk utiliza ciertas secuencias de caracteres, llamadas secuencias de escape, para especificar caracteres para los cuales no existe otra notación. Por ejemplo, `\n`, significa un carácter de nueva línea, el cual no puede aparecer representado de otra manera en un string o en una expresión regular; `\t` significa tabspace, `\r` significa un tabulador, `\f` significa formfeed, `\v` significa el carriage return, y `\c`, donde c es literalmente cualquier otro carácter (`\` para backslash, `"` para las comillas, `V` para un slash).

Las secuencias de escape tienen significado especial solamente dentro de los programas awk; en los datos no son más que simples caracteres.

3.4.5 PATRONES COMPUESTOS

Un patrón compuesto es una expresión que combina patrones, usando paréntesis y los operadores lógicos || (OR), && (AND), y ! (NOT).

Un patrón compuesto equipara la línea de entrada actual si la expresión evaluada es verdad.

El operador ! (NOT) tiene mayor precedencia que && (AND). El operador && (AND) tiene mayor precedencia que || (OR).

Los operadores && y || evalúan sus operandos de izquierda a derecha. La evaluación termina apenas se ha determinado la verdad o falsedad de la expresión.

3.4.6 RANGO DE PATRONES

Un rango de patrones consiste de dos patrones separados por una coma, como en patr1, patr2.

Un rango de patrones equipara cada línea entre una ocurrencia de patr1 y la siguiente ocurrencia de patr2 inclusive.

La equiparación empieza siempre que el primer patrón de un rango equipare.

Si no se encuentra una instancia del segundo patrón, entonces todas las líneas desde que equiparó el primer patrón hasta el final de la entrada son equiparadas.
Un rango de patrones no puede ser parte de otro patrón.

3.4.7 RESUMEN DE PATRONES

v BEGIN { sentencias }
Las sentencias son ejecutadas una sola vez antes de que hayan sido leídas las líneas de entrada.

v END { sentencias }
Las sentencias son ejecutadas una vez después de que todas las líneas de entrada han sido leídas.

v expresión { sentencias }
Las sentencias son ejecutadas sobre cada línea de entrada donde la expresión es verdad, esto es un valor diferente de cero o no nula.

v /expresión regular/ { sentencias }
Las sentencias son ejecutadas sobre cada línea de entrada que contiene un string que es equiparado por la expresión regular.

v padrón compuesto { sentencias }
Un patrón compuesto combina expresiones con && (AND), || (OR), ! (NOT) y paréntesis; las sentencias son ejecutadas en cada línea de entrada en las cuales el patrón compuesto es verdad.

v patron1, patron2 { sentencias }
Una rango de patrones equipara cada línea de entrada desde una línea equiparada por patrón1 hasta la siguiente línea equiparada por patrón2, inclusive; las sentencias son ejecutadas sobre cada línea que equipara.

Notas:
- BEGIN y END no se combinan con otros patrones.
- Un rango de patrones no puede ser parte de otro patrón.
- BEGIN y END son los únicos patrones que obligadamente requieren una acción.

3.5 ACCIONES

En una sentencia patrón-acción, los patrones determinan cuando las acciones deben ser ejecutadas.

En ocasiones, una acción es muy simple: un print o una asignación. Otras veces, puede ser una secuencia de muchas sentencias separadas por nueva línea (\n) o por punto y coma (;).

UNIX: Una puerta abierta a los sistemas abiertos
3.5.1 CONSTANTES
Existen dos tipos de constantes las cuales pueden ser identificadas como numéricas y string.

3.5.1.1 Constantes Numéricas
Una constante numérica puede ser un entero (ejemplo 230), un número decimal (ejemplo, 123.34), o un número en notación científica (ejemplo, 0.36E-12).

Diferentes representaciones de un mismo número tienen el mismo valor numérico.

Todos los números se almacenan en coma flotante, cuya precisión es dependiente de máquina.

3.5.1.2 Constante string
Una constante string es una secuencia de cero o más caracteres encerrados entre comillas. Ejemplo: "ab", "ab".

Una comilla puede ser parte del string si va precedida de \. Si se desea poner el caracter de nueva línea en el string, se puede usar \n.

Ejemplo: "La \"historia\" del tiempo\n".

3.5.2 VARIABLES
Las expresiones pueden contener varios tipos de variables: definidas por el usuario, interconstruidas, y campos.

Los nombres de variables definidas por el usuario son secuencias de letras, dígitos y caracteres de subrayado (_), que no empiecen con un dígito.

Todas las variables interconstruidas tienen nombres en mayúsculas.

Una variable tiene un valor que es un string, un número, o los dos.

Ya que el tipo de dato que va a contener la variable no se declara, awk infiere el tipo del contexto en que se presenta la variable.

Cuando sea necesario, awk convertirá un valor string en un valor numérico, o vise versa.

Una variable no inicializada toma por omisión el valor string "" (string nulo) y el valor numérico 0 (cero).

3.5.2.1 Variables inter-construidas
A continuación se especifican las variable inter-construidas existentes en awk:

| v | ARGC | número de argumentos en la línea de comandos |
Las variables mencionadas pueden ser usadas en cualquier expresión; y sus valores, asignados por el usuario.

FILENAME es inicializada cada vez que un nuevo fichero se lee.

FNR, NF, y NR toman un valor cada vez que un registro es leído; adicionalmente NF toma un nuevo valor cuando el registro cambia o un nuevo campo se ha creado.

RLENGTH y RSTART cambian como resultado de invocar a la función match.

NF (número de campos), es una variable especial cuyo valor es el número de campos en el registro de entrada actual.

NR (número de registros) es una variable cuyo valor es el número de registros leídos. El primer registro leído es el 1.

RS (separador de registro) es una variable cuyo valor es el separador actual entre registros. El valor de RS inicialmente está inicializado por omisión a nueva línea y, si se desea, puede ser cambiado a cualquier carácter c al incluir la sentencia de asignación RS = "c" en una acción.

FS (separador de campo) es una variable que indica el separador de campo actual. Inicialmente, el valor de FS por omisión es un blanco, indicando que los campos son separados por espacio en blanco (cualquier secuencia no nula de blancos y tabuladores). El valor de FS puede ser cambiado a cualquier carácter simple c al incluir la sentencia de asignación FS = "c" en una acción, o usando el argumento opcional -Fc (c es el carácter) en la línea de comandos. Dos valores de o tienen significado especial, espacio y t. La sentencia de asignación FS = " " hace que el carácter separador de campo sea un espacio en blanco; y en la línea de comandos, -Ft hace que el separador de campos sea un tabulador.
OFS (separador de campo de salida) es el separador de campo para la salida. Es colocada entre campos para impresión.

El valor de ORS (separador de registros de salida) es puesto después de cada registro para impresión.

Por omisión, OFS es inicializado a espacio y ORS a nueva línea. Estos valores pueden cambiar a cualquier string por asignaciones tales como: ORS = "abc" y OFS = "xyz".

3.5.2.2 Variables de campo

$0 (cero) es una variable cuyo valor es el de la línea de entrada actual (se refiere a toda la línea).

Los campos de la línea de entrada actual son llamados $1, $2, ..., hasta $NF.

Así $NF tiene, como valor, el valor del último campo del registro de entrada actual.

Los campos comparten las propiedades de las otras variables: pueden recibir asignaciones y ser usadas en operaciones aritméticas y de strings.

Los campos pueden ser especificados por expresiones. Por ejemplo, $(NF-1) es el penúltimo campo de la línea de entrada. Los paréntesis son necesarios ya que $NF-1 significa el valor numérico del último campo menos uno.

Una variable de campo que hace referencia a un campo no existente, por ejemplo, $(NF+1) tiene como valor inicial el valor del string nulo. Se puede crear un nuevo campo asignándole un valor. Ejemplo, si el registro tiene únicamente cuatro campos, el programa siguiente caerá un adicional:

```
{ $5 = 1000 * $3 / $2
  print $5 }
```

El número de campos puede variar de línea a línea, pero existe una limitación en la implementación de hasta 100 campos en un registro.

3.5.3 OPERADORES

Awk tiene operadores de asignación, aritméticos, relacionales y lógicos, similares a los utilizados en el lenguaje C. Además cuenta con operadores para equiparación de patrones similares a aquellos de los programas lex y grep del sistema operativo UNIX.

3.5.3.1 Operadores aritméticos

Awk provee los operadores aritméticos usuales: +, -, *, /, %, y ^.

El operador % calcula el resto de la división: x%y es el resto cuando x es dividido por y. Ejemplo 12 % 2 es 1. Su comportamiento, si x o y son números negativos, depende de la máquina.

El operador ^ es de exponenciación: x^y es x elevado a y.
Toda la aritmética es efectuada en coma flotante.

3.5.3.2 Operadores de comparación

Las expresiones de comparación son aquellas que contienen, sea un operador relaciona1 o, un operador para equiparación de expresiones regulares.

Los operadores relacionales son:

- `v <` menor que
- `v <=` menor o igual que
- `v ==` igual que
- `v !=` no es igual que
- `v >=` mayor o igual que
- `v >` mayor que

Los operadores para equiparación de expresiones regulares son:

- `v ~` equiparado por
- `v !~` no equiparado por

Ejemplo:

```
$4 -/Asia/
```

es verdad si el cuarto campo de la línea de entrada actual contiene Asia como un substring del mismo.

3.5.3.3 Operadores lógicos

Los operadores lógicos `&&` (AND), `||` (OR), y `!` (NOT) son usados para crear expresiones lógicas por combinación con otras.

Los operandos de expresiones separadas por `&&`, `||` o `!`, son evaluados de izquierda a derecha, y la evaluación cesa el momento en que el valor de la expresión completa puede ser calculado. Ejemplo `expre1 && expre2`, `expre2` no es evaluada si `expr1` es falso, mientras que en `expr3 || expr4`, `expre4` no es evaluada si `expre3` es verdad.

3.5.3.4 Operadores de incremento y decremento

Los operadores de incremento y decremento son `++` y `--`, respectivamente; y los dos tienen formas prefija y postfija de operación.

```
n = n + 1
```

se escribe usualmente `n++` (incremento postfijo) o `++n` (incremento prefijo).

UNIX: Una puerta abierta a los sistemas abiertos
La forma prefija incrementa 1 a n antes de "utilizar o deliberar" el valor de n; la forma postfija incrementa 1 a n después de "utilizar o deliberar" el valor de n. Esto marca diferencias en el momento de utilizar ++ en una asignación.

Ejemplo: si n inicialmente vale 1, i = ++n, incrementa n 2 a i; al contrario, i = n++, asigna el valor 1 a i e incrementa n a 2. En los dos casos, n queda incrementada en uno su valor, sin embargo la asignación a i varía.

Si lo que se quiere es solamente incrementar n, entonces no existe diferencia en utilizar n++ o ++n.

El operador de decremento (--), postfijo y prefijo, el cual substraer uno de la variable, trabaja de manera similar.

3.5.3.5 Operadores de asignación

Existen siete operadores de asignación que pueden ser usados en expresiones denominadas asignaciones.

La asignación más simple es una expresión de la forma

\[ \text{var} = \text{expr} \]

donde var es el nombre de una variable o un campo, y expr es una expresión.

Los otros seis operadores de asignación son: + =, - =, * =, / =, % = y ^ =. Sus significados son similares: var op = expr tiene el mismo significado que \[ \text{var} = \text{var} \text{op} \text{expr} \], excepto que var es evaluada solamente una vez. Ejemplos, \[ x = x + 3 \] es lo mismo que \[ x += 3 \].

Note que una asignación es una expresión; y su valor es el nuevo valor del lado izquierdo de la asignación. Así, la asignación puede ser usada sobre cualquier expresión. Ejemplo, una múltiple asignación, \[ FS = OFS = "\\t" \], en la que tanto al separador de campo de entrada como al separador de campo de salida se les asigna el tabulador.

3.5.3.6 Operadores de string

Existe solamente una operación de strings, la concatenación. La concatenación no tiene un operador explícito: las expresiones string se forman al escribir constantes, variables, campos, elementos de un arreglo, y otras expresiones, una a continuación de otra. Ejemplo: el programa

```plaintext
{ print NR ":= " $0 }
```

escribe cada línea precedida por su número de línea y dos puntos. El número NR es convertido a un valor string (y también $0, si es necesario): entonces los tres strings son concatenados y el resultado es impreso.

3.5.3.7 Funciones aritméticas inter-construidas

Las funciones aritméticas inter-construidas son las siguientes:
\[
\begin{align*}
\text{atan2}(y,x) & \quad \text{arctangente de } y/x \text{ en el rango } -\pi \text{ a } +\pi \\
\cos(x) & \quad \text{coseno de } x, \text{ con } x \text{ en radianes} \\
\exp(x) & \quad \text{función exponencial de } x, \text{ e a la } x \\
\int(x) & \quad \text{parte entera de } x \\
\log(x) & \quad \text{logaritmo natural de } x \text{ (base e)} \\
\rand() & \quad \text{número aleatorio } r, \text{ donde } 0 \leq r \leq 1 \\
\sin(x) & \quad \text{seno de } x, \text{ con } x \text{ en radianes} \\
\sqrt(x) & \quad \text{raíz cuadrada de } x \\
\srand(x) & \quad x \text{ es punto de partida para } \rand() \\
\text{Nota: Para calcular algoritmos en base } 10 \text{ de } x, \text{ use } \log(x)/\log(10)
\end{align*}
\]

3.5.3.8 Strings como expresiones regulares

Una expresión regular puede ser utilizada como el lado derecho de los operadores ~ y !~.

\textit{Ejemplo: el programa}:

\begin{verbatim}
BEGIN { digits = "^[0-9]+$" }
$2 ~ digits

imprime todas las líneas en las cuales el segundo campo es un string de dígitos.
\end{verbatim}

Debido a que las expresiones pueden ser concatenadas, una expresión regular puede ser construida a partir de componentes.

\Rightarrow \quad \text{El siguiente programa imprime las líneas de entrada que son números de coma flotante válidos:}

\begin{verbatim}
BEGIN

sign = "[+-]"
decimal = "[0-9]+[.][0-9]" 
fraction = ".[0-9]+"
exponente = "([eE]) sign [0-9]"

number = "^" sign "( decimal | fraction )" exponent "$"

$0 ~ number
\end{verbatim}

3.5.3.9 Funciones string inter-construidas

Awk provee las siguientes funciones string inter-construidas:

\textit{UNIX: Una puerta abierta a los sistemas abiertos}
index(s,t)
Retorna la primera posición donde el string t empieza en s, o 0 (cero) si t no está presente en s. El primer carácter en un string está en la posición 1.

Ejemplo:
index("banana","an")

Esta función retorna el valor 2.

length(s)
Retorna el número de caracteres en s.

match(s,r)
Encuentra el primer substring desde la izquierda en el string s que es equiparado por r. Retorna el índice en donde se encuentra el substring, ó 0 (cero) si no se encuentra en s. Además, en RSTART coloca el valor del índice encontrado y en RLENGTH coloca la longitud del substring equiparado.

split(s,a)
Parte s en un arreglo denominado a de acuerdo a FS, retorna el número de campos.

split(s,a,fs)
Parte s en un arreglo denominado a de acuerdo al separador de campos especificado en fs; retorna el número de campos.

sprintf(formato,expr1,expr2,...,exprn)
Retorna (sin imprimir) un string que contiene expr1, expr2, ..., exprn, formateado de acuerdo a lo especificado en el valor string de la expresión formato.

Ejemplo:

x = sprintf("%10s %6d", $1, $2)
asigna a x el string producido al formatear los valores de $1 y $2 como un string de 10 caracteres y como un número decimal en un campo de almenos seis posiciones.

sub(r,s,t)
Encuentra el mayor substring de t, empezando desde la izquierda, equiparado por la expresión regular r; luego reemplaza el substring encontrado por el string de sustitución s. Retorna el número de sustituciones efectuadas.

gsub(r,s,t)
Esta función es similar a sub(r,s,t), excepto que sucesivamente sustituye s por los substring no sobrepuestos equiparados por r en el
string t (la g indica global). Retorna el número de sustituciones efectuadas.

Ejemplo:

gsub(/ana/, "anda", "banana")

reemplazará banana por bandana; note que banana tiene dos substring “ana”, pero el segundo está sobrepuesto en el primero (“anana”), por lo que no se equipará el segundo para reemplazo por el string de sustitución s (en este caso "anda").

v sub(r,s)

Es el sinónimo de sub(r,s,$0). Trabaja en forma similar a sub(r,s,t), con la diferencia que el string en donde buscar es $0 (la línea de entrada actual) y no t.

v gsub(r,s)

Esta función es sinónimo gsub(r,s,$0).

Ejemplo:

{ gsub(/UIO/, "Quito")
  print }

transcribirá la entrada, reemplazando todas las ocurrencias de "UIO" por "Quito".

v substr(s,p)

Retorna el sufijo de s que empieza en la posición p.

v substr(s,p,n)

Retorna el substring de s de longitud n empezando en la posición p.

Ejemplo: si $0 es "Ecuador país de paz"

{ $1 = substr($1,1,3) ; print $0 }

producirá: "Ecu país de paz"

Nota: En los parámetros de las funciones y en la descripción de las mismas, r representa una expresión regular (sea como un string o delimitado por slashes (\ )); s y t son expresiones string, en tanto que n y p son enteros.

3.5.4 EXPRESIONES

Las expresiones son las sentencias más simples. Una expresión se forma al combinar expresiones primarias (y otras) con operadores.

Las expresiones primarias son los "building blocks"; incluyen constantes, variables, referencias a arreglos, invocación a funciones, y varias construcciones propias (ejemplo: funciones interconstruidas).
3.5.4.1 Expresiones condicionales

Una expresión condicional tiene la forma que se muestra a continuación:

\[ \text{expr1 ? expr2 : expr3} \]

Primeramente, expr1 se evalúa. Si es verdad, esto es un valor diferente de cero o de nulo, el valor de la expresión condicional es el valor de la expr2. De otra manera, es el valor de expr3.

Solamente una expresión es evaluada para el resultado, expr2 o expr3.

3.6 ENTRADA

Existen muchas maneras en las que se puede proveer entrada a un programa awk. La forma más sencilla es poner los datos de entrada en un fichero, digamos "datos", y luego digitar

\[ \text{awk \'programa\' datos} \]

Awk lee de la entrada standard si no se ha indicado ningún nombre de fichero; de esa manera, una segunda manera de proveer entrada es tomarla de la salida de un pipe.

3.6.1 SEPARADORES DE ENTRADA

Las variables FS y RS contienen el separador de campo de entrada y el separador de registro de entrada, respectivamente.

3.6.2 REGISTROS Y CAMPOS

Awk lee como entrada un registro a la vez. Un registro es una secuencia de caracteres que terminan con una nueva línea (a no ser que se haya especificado lo contrario en la variable RS) o con fin de fichero. Así el registro es una línea de entrada.

El string de caracteres, así leídos, se asignado a la variable $0.

Una vez que awk a leído un registro, éste es dividido en campos. Y cada campo dentro de un registro es asignado a las variables $1, $2, ..., de acuerdo al número de campos que tenga un registro en particular. Un campo es una secuencia de caracteres separados por blancos o tabuladores (al no ser que se haya especificado lo contrario en la variable FS).

3.6.3 LÍNEA DE COMANDOS

En awk no se declaran tipos de variables, una variable es creada al ser referenciada. Es posible asignar valores a una variable que es referenciada en un programa awk.

Ejemplo:

\[ x = 5, \text{ asignará 5 a la variable } x. \]
También es posible asignar valores a las variables desde la línea de comandos. Esto provee otra manera de suplir valores de entrada a un programa awk.

3.7 SALIDA

3.7.1 LA SENTENCIA PRINT

La sentencia print tiene dos formas, las cuales pueden ser representadas como sigue:

\[
\text{print } \text{expr1, expr2, ..., exprn} \\
\text{print (expr1, expr2, ..., exprn)}
\]

Las dos formas imprime las expresiones separadas por OFS (separador de campo de salida) y terminadas por ORS (separador de registro de salida).

Ejemplo, el programa:

\[
\begin{align*}
\text{BEGIN} & \{ \text{OFS } = \\
\text{ } & \text{"\text{"} } \\
\text{ } & \{ \text{print } \$1, \$2, \$3 \} \\
\text{ } & \{ \text{print } \$1 \$2 \$3 \}
\end{align*}
\]

para los registros de entrada:

Ingeniería de Sistemas
Escuela Politécnica Nacional

producirá la siguiente salida:

Ingeniería de Sistemas
Ingeniería de Sistemas
Escuela Politécnica Nacional

Ya que \$1, \$2, \$3 en la segunda sentencia patrón-acción (la primera es la del BEGIN) son expresiones que al ser impresas las separa el OFS; en tanto que \$1 \$2 \$3 en la tercera sentencia patrón-acción constituye una sola expresión string que resulta de la concatenación de \$1 \$2 \$3.

La sentencia print sin expresiones a su derecha, es una abreviación de print $0 (imprime todo el registro leído).

Para imprimir una línea en blanco, esto es, una línea con solamente un nueva línea, use print "."

La segunda forma de la sentencia print, encierra la lista de argumentos entre paréntesis, como en:

\[
\text{print (} \text{expr1, expr2, ..., exprn)}
\]

UNIX: Una puerta abierta a los sistemas abiertos
Las dos formas de la sentencia print generan la misma salida, excepto que los paréntesis son necesarios para argumentos que contienen operadores relacionales.

### 3.7.2 SEPARADORES DE SALIDA

Las variables OFS y ORS contienen el separador de campo de salida y el separador de registro de salida, respectivamente.

### 3.7.3 LA SENTENCIA PRNNTF

La sentencia printf es usada para generar salida formateada, y es muy similar a la sentencia printf utilizada en C, excepto que el especificador de formato * no es soportado.

Tiene dos formas, similares a las de print:

- `printf formato, expr1, expr2, ..., exprn`
- `printf (formato, expr1, expr2, ..., exprn)`

El argumento formato debe estar presente siempre, y es una expresión cuyo valor string contiene texto literal a ser impreso, especificaciones de como las expresiones (expr1, ..., exprn) serán formateadas, ó texto y especificaciones a la vez.

Cada especificación de formato empieza con % y termina con un carácter que determina la conversión. Ejemplos:

<table>
<thead>
<tr>
<th>Caracter</th>
<th>Imprima la expresión como</th>
</tr>
</thead>
<tbody>
<tr>
<td>c</td>
<td>carácter ASCII</td>
</tr>
<tr>
<td>d</td>
<td>entero decimal</td>
</tr>
<tr>
<td>e</td>
<td>notación exponencial ([+-]d.ddddddE[+-+]dd)</td>
</tr>
<tr>
<td>f</td>
<td>número real ([+-]ddd.dddddd)</td>
</tr>
<tr>
<td>s</td>
<td>string</td>
</tr>
</tbody>
</table>

Entre el % y el carácter es posible incluir tres modificadores:

- `-` : expresión justificada a la izquierda.
- `ancho` : ancho del campo.
- `precisión ` : máximo ancho de string o, dígitos a la derecha del punto decimal.

Ejemplos de especificaciones de impresión:

<table>
<thead>
<tr>
<th>formato</th>
<th>$1</th>
<th>printf(formato,$1)</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;%c&quot;</td>
<td>97</td>
<td>a</td>
</tr>
<tr>
<td>&quot;%d&quot;</td>
<td>97.5</td>
<td>97</td>
</tr>
<tr>
<td>&quot;x = %f&quot;</td>
<td>97.5</td>
<td>x = 97.500000</td>
</tr>
</tbody>
</table>

UNIX: Una puerta abierta a los sistemas abiertos
"x = %7.2f" 97.5  x = 97.50
"%s" Enero |Enero|
"%10s" Enero | Enero|
"%-10s" Enero |Enero |
"%-10.3s" Enero | Ene|
"%-10.3s" Enero |Ene |

3.7.4 SALIDA EN FICHeros

Los operadores de redirección > y >> pueden ser utilizados para dirigir la salida a ficheros en vez de a la salida estándar.

El siguiente programa pondrá el primero y el tercer campo de todas las líneas de entrada en dos ficheros: mayores si el tercer campo es mayor que 10 y menores en caso contrario.

\$3 > 18 \{ print $1, $3 > "mayores" \}
\$3 <= 18 \{ print $1, $3 > "menores" \}

Note que los nombres de fichero tienen que estar delimitados por comillas ("); sin las comillas, mayores y menores son meramente variables no inicializadas.

Los nombres de fichero pueden ser variables o expresiones, como en:

\{ print($1, $3) > ($3 > 100 ? "mayores": "menores") \}

hace lo mismo que el programa anterior.

En tanto que en:

\{ print > $1 \}
imprime cada línea de entrada en un fichero nombrado en el primer campo.

Nota: Tanto en print como en printf, si una expresión en la lista de argumentos a imprimirse (expr1, ..., exprn) contiene un operador relacional, entonces sea la expresión o la lista de argumentos deben ir entre paréntesis.

La regla anterior elimina cualquier posibilidad de ambigüedad al utilizar el operador > de redirección de salida.

En:

\{ print $1, $2 > $3 \}

> es el operador de redirección, y no es parte de la segunda expresión; así, los valores del primero y segundo campos son escritos en el fichero nombrado por en el tercer campo.

Si se desea que la segunda expresión incluya el operador > (mayor que), utilice los paréntesis:
Es importante notar que un operador de redirección abre un fichero solamente una vez. Cada print o printf sucesivos añaden más datos al fichero abierto.

Sin embargo, cuando se utiliza el operador de redirección >, el fichero es primeramente aclarado antes de que se escriba sobre él. En tanto que, si utilizarnos >> en vez de >, el fichero no es aclarado; la salida es anadida a continuación del contenido original.

3.7.5 **SALIDA A "PIPES"**

Es posible direccionar la salida a un "pipe" en vez de hacerlo a un fichero (en un sistema que soporte "pipes").

La sentencia:

```
{print / comando}
```

hace que la salida de print sea la entrada de comando.

Cierre de ficheros y "pipes"

La sentencia close(expr) cierra un fichero o un "pipe" denotado por expr. El valor del string expr debe ser el mismo que el valor del string que fue usado en primera instancia para crear un fichero o un "pipe". Ejemplo:

```
close("mayores")
```

cerrará el fichero denominado mayores.

Close es necesario si lo que se intenta es escribir en un fichero y luego leerlo en el mismo programa.

El límite del número de ficheros abiertos al mismo tiempo es dependiente del sistema en el que se esté trabajando.
4.1 ANALIZADOR LEXICO: LEX

Lex es un programa generador que produce, en algún lenguaje de propósito general, un programa que reconoce expresiones regulares. Su diseño se orienta al procesamiento léxico de cadenas de caracteres de entrada. Las expresiones regulares son especificadas por el usuario en las especificaciones fuente dadas a Lex.

Un programa fuente en Lex (programa generador) es una tabla de expresiones regulares y sus correspondientes fragmentos de programa. A partir de esta tabla se produce un programa que se encarga de leer una cadena de entrada, copiarla en una cadena de salida y dividir la entrada en strings que empaten con las expresiones dadas. Se ejecuta un fragmento de programa con cada string reconocido, y su orden estará dado según la cadena de entrada. Lex genera un autómata finito determinístico que efectúa el reconocimiento de las expresiones.

Muy a menudo se realiza una combinación de Lex y Yacc: esta mezcla es muy apropiada, para ejecutar la fase de análisis léxico, ya que el programa de lex reconoce solo expresiones regulares, en tanto que yacc escribe parsers que aceptan una gran clase de gramáticas de contexto libre pero necesita de un analizador de bajo nivel para reconocer los tokens de entrada. De esta forma lex es utilizado como un preprocesador.
4.1.1 SECCIONES DE UN PROGRAMA EN LEX

De manera general Lex presenta las siguientes secciones en un fichero fuente:

```
{ definiciones }
%
{ reglas }
%
( subrutinas de usuario )
```

Tanto las definiciones como las subrutinas de usuario, son a menudo omitidas. Los primeros `%%` indican el inicio de las reglas y son obligatorios; los segundos `%%` son opcionales, por lo cual el más pequeño programa en Lex será:

```
%%
es decir, sin regla ni definiciones y que será traducido en un programa que copiará una entrada a una salida sin ningún cambio.
```

La sección de definiciones contiene una combinación de:

- Definiciones en la forma "traducción de nombre de espacio"
- Código incluido en la forma "código de espacio"
- Código incluido en la forma:
  ```
  % {
  code
  % }
  ```
- Condiciones iniciales dadas en la forma:
  ```
  %S nombre1 nombre2 ... 
  ```
- Tablas de conjuntos de caracteres en la forma:
  ```
  % T
  numero de espacio de caracter-string
  ...
  % T
  ```
- Cambios a tamaños internos de arreglos en la forma:
  ```
  %x nnn
  ``
  donde nnn es un entero decimal representando un tamaño de arreglo.

Las reglas representan decisiones de control y se encuentran en una tabla que contiene:

- Una columna izquierda con expresiones regulares, y
Una columna derecha con acciones y fragmentos de programas a ser ejecutados cuando las expresiones sean reconocidas.

**Ejemplo:**

```plaintext
real       printf(" reconocido REAL ")
hola       printf(" HOLA ")
```

Las subrutinas de usuario, serán cualquier tipo de subrutinas escritas en lenguaje C.

### 4.1.2 SINOPSIS

`lex [ -rctvn ] [ file ] ...`

#### 4.1.2.1 Opciones

Las opciones deben aparecer antes de los ficheros fuente en la línea de comandos. Múltiples ficheros son tratados como un solo fichero. Si no se especifica ningún fichero, la entrada estándar es usada.

- `-r` indica acciones RATFOR.
- `-c` indica acciones C (por omisión).
- `-t` escribe el programa lex.yy.c a la salida estándar.
- `-v` provee un sumario de los datos estadísticos generados de la máquina.
- `-n` suprime el sumario que genera `-v` (por omisión).

**Nota:** La opción `-r` no es totalmente funcional.

### 4.1.3 FORMA DE COMPILACION

Un programa fuente en Lex, es compilado en dos pasos, los cuales se describen a continuación:

- **a)** El fuente en Lex es cambiado en un programa que se genera en el lenguaje de propósito general huésped del sistema,

- **b)** El programa generado debe ser compilado y cargado usualmente con una librería de las subrutinas Lex.

El programa generado es un fichero llamado lex.yy.c, que como ya se dijo, reconoce expresiones regulares y ejecuta las acciones correspondientes indicadas por el usuario.

En el sistema operativo UNIX, la librería es accesaada por la bandera `-l`.

**Ejemplo:**

```plaintext
lex fuente       {produce el fichero lex.yy.c}
cc lex.yy.c -l   {compila el fichero lex.yy.c}
```
El resultado de la compilación del fichero lex.yy.c es ubicado en el conocido fichero a.out para su posterior ejecución.

### 4.1.4 NOTACIÓN

Tanto en la sección de definiciones como en la de subrutinas de usuario se emplea la misma notación de lenguaje C. En la sección de reglas, para la definición de las expresiones regulares en Lex, se utiliza la siguiente notación.

- **x** el carácter "x".
- "x" una "x", solo si x es un operador.
- \x una "x", solo si x es un operador.
- [xy] el carácter x o y.
- [x-z] el carácter x, y o z.
- [^x] cualquier carácter excepto x.
- . cualquier carácter excepto nueva línea.
- ^x una x en el inicio de una línea.
- <y>x una x cuando Lex está en la condición inicial y.
- x$ una x en el fin de una línea.
- x? una x opcional.
- x* 0,1,2,... instancias de x.
- x+ 1,2,3,... instancias de x.
- x|y una x o una y.
- (x) una x.
- x/y una x pero solo si está seguida por y.
- {xx} la traducción de xx desde la sección de definiciones.
- x{m,n} de m a n ocurrencias de x.

**Ejemplo:**

```c
int vector[100];

%(a-z)+ vector[yylen]+++;

. |
\n;

%%

yywrap()
```
int i;
printf("Longitud Número de palabras\n");
for (i=0;i<100;i++)
  if (vector[i]>0)
    printf("%5d%10d\n", i, vector[i]);
return(1);
}

Para finalizar, podemos describir a continuación una serie de características importantes de lex:

- Lex es un programa generador que produce, en algún lenguaje de propósito general, un programa que reconoce expresiones regulares. Su diseño se orienta al procesamiento léxico de cadenas de caracteres de entrada.
- Las expresiones regulares que entran a Lex son convertidas a máquina de estado finito mediante la producción de una tabla de transición, la cual se genera automáticamente.
- El fichero que obtenemos después de la compilación con lex (yy.tab.o), contendrá la tabla de transición más las acciones a seguirse, escritas en lenguaje C.
- El fichero en C obtenido (yy.tab.o) al ser compilado, producirá un ejecutable que funcionará como un scanner, el mismo que producirá tokens de entrada al parser.
- En un programa en lex se puede reemplazar expresiones regulares por nombres, determinar acciones para ciertos patrones mediante rutinas en lenguaje C y la codificación de las mismas.

4.2 ANALIZADOR SINTACTICO: YACC

Siglas de yet another compiler-compiler. Yacc toma una gramática de contexto libre y la convierte en un conjunto de tablas para un automata simple, el cual ejecuta un algoritmo de parsing de gramáticas LR(1). La gramática puede ser ambigua, puesto que, yacc usa reglas de precedencia especificadas para eliminar ambigüedades.

Dicho en forma general, el programa yacc genera una función para controlar el proceso de entrada. Esta función denominada parser, llama a una analizador léxico que toma los items básicos, conocidos como tokens, de la cadena de entrada. Los tokens se organizan de acuerdo a reglas gramaticales, las cuales al ser reconocidas invocan a un código de usuario.

El parser producido por yacc consiste de una máquina de estado finito con una pila. El parser es también capaz de leer y recordar el siguiente token de entrada, conocido como token look-ahead. El estado actual es siempre aquel que está en el tope de la pila. Los estados de la máquina de estado finito son
representados por enteros; inicialmente se tiene el estado 0 y ningún token look-ahead ha sido leído. La máquina dispone de cuatro acciones: shift, reduce, accept y error.

4.2.1 SECCIONES DE UN PROGRAMA EN YACC

Cada fichero de especificación consiste de tres secciones: las declaraciones, reglas (gramaticales) y programas. Entonces la estructura básica será:

```
declaraciones
%%
reglas
%%
programas
```

En la sección de declaraciones se declaran los tokens a ser utilizados y organizados por el parser. Además de los tokens también pueden declararse variables e incluir programas externos.

En la sección de reglas, se escribe las reglas gramaticales que organizarán a los tokens.

En la sección de programas se puede escribir la rutina de análisis léxico, además de otro tipo de programas que el usuario considere necesarios.

Los primeros %% son obligatorios. La sección de declaraciones puede estar vacía, y si la sección de programas es omitida, los segundos %% también pueden ser omitidos. Por tanto, la más pequeña especificación en YACC podría ser:

```
%%
reglas
```

4.2.2 SINOPSIS

```
yacc [-vdlt] gramática
```

4.2.2.1 Opciones

- **-v** prepara el fichero y.output.
- **-d** genera el fichero y.tab.h.
- **-l** produce código en y.tab.c, el cual no contiene alguna construcción #line. Esta opción deberá ser usada solo después de la gramática y las acciones asociadas son totalmente depuradas.
- **-t** incluye código de depuración en tiempo de ejecución cuando y.tab.c es compilado.

UNIX: Una puerta abierta a los sistemas abiertos
4.2.3 FORMA DE COMPILACION

De acuerdo a las opciones indicadas, al ser compilada la especificación fuente en yacc se producirán algunos ficheros. Cuando no se indica ninguna opción, se produce el fichero y.tab.c, que posteriormente deberá ser pasado por el compilador de lenguaje C.

Los ficheros obtenidos son:

- y.tab.c: contiene el analizador sintáctico escrito en lenguaje C.
- y.output: describe las tablas de parsing y reporta conflictos generados por ambigüedades en la gramática. Genera el código del automata finito.
- y.tab.h: contiene las sentencias #define que asocian los códigos de token asignados por yacc con los nombres de token declarados por el usuario. Esta opción permite a otros ficheros fuente como y.tab.c accesar los códigos de los token.

Ejemplo: Si un fichero de yacc se llama ejem1, entonces formas de compilatorio serán:

```
yacc ejem1         {genera fichero y.tab.c}
yacc -v ejem1      {genera fichero y.output}
yacc -dv ejem1     {genera fichero y.tab.h}
```

4.2.4 NOTACIÓN

En general yacc emplea la misma notación que el lenguaje C, excepto para la declaración de tokens y la definición de reglas gramaticales.

```
%          inicio de declaración de token
%%         separador de secciones
a:b        se lee, "a” se define como ”b"
;           indica fin de regla gramatical
```

Un token se declara de la siguiente forma:

```
%token nombre1 nombre2
```

Cada nombre no definido en la sección de declaraciones es asumido que representa un símbolo no terminal. Cada símbolo no terminal debe aparecer en el lado izquierdo de al menos una regla.

Una regla gramatical tiene la forma:

```
A: CUERPO ;
```
donde "A" representa un nombre de un no terminal, y "BODY" representa una secuencia de cero o más nombre y literales. Mayúsculas y minúsculas son distintas. Los nombres usados en el cuerpo de una regla gramatical pueden representar tokens o símbolos no terminales. Si algunas reglas gramaticales tienen el mismo lado izquierdo, se puede utilizar la barra vertical para evitar el escribir nuevamente este lado, es decir:

A:B;
A:C;
A:D;

puede ser escrito como:

A: B 
| C 
| D;

Ejemplo:
%token identifier
%%
expression
  : expression / identifier
   / identifier
   ;

Para finalizar, podemos describir a continuación una serie de características importantes sobre el yacc:

- Yacc toma una gramática de contexto libre y la convierte en un conjunto de tablas para un autómata simple.
- Yacc usa reglas de precedencia especificadas para eliminar ambigüedades de las gramáticas.
- Yacc genera una función denominada parser y llama a una analizador léxico que toma los items básicos conocidos como tokens de la cadena de entrada. Los tokens se organizan de acuerdo a reglas gramaticales, las cuales al ser reconocidas invocan a un código de usuario.
- El parser producido por un programa en yacc consiste en una máquina de estado finito.

UNIX: Una puerta abierta a los sistemas abiertos
v El programa en yacc al ser compilado produce un fichero (y.tab.c) que corresponde al analizador sintáctico en lenguaje C.

v La combinación de Lex y Yacc es muy apropiada, ya que el programa de lex reconoce expresiones regulares, en tanto que yacc escribe parsers que aceptan una gran clase de gramáticas de contexto libre.

4.3 EXPRECIIONES REGULARES

A continuación presentamos una breve descripción de expresiones regulares muy útiles (/, => y []). Cualquier información adicional sobre este tópico, se encontrará más detallada en el manual.

a) Expresión regular "/": Indica un tipo de condición necesaria, en donde el carácter a la izquierda de "/" será reconocido solo si el carácter a la derecha de "/" está presente.

Ejemplo:

x/y --> una x solo si está seguida de y.

b) Expresión regular "[]": Puede indicar una de dos cosas, para esto se explicará mediante la ayuda de dos ejemplos:

Ejemplo:

{x} --> indica la traducción de xx desde la sección de definiciones.

x{m,n} --> indica el número de ocurrencias (de m a n) del carácter x.

c) Expresión regular "<>": Indica un tipo de condición inicial de lex, para reconocer un carácter determinado.

Ejemplo:

<y>x --> una x cuando lex está en la condición inicial y.

4.4 PROGRAMAS EJEMPLO

a) Este programa en lex realiza la impresión de toda cadena ingresada con excepción de los caracteres 'blanco', 'tab' y 'new line'. Esto se explica debido que para los caracteres 'blanco'(' '), 'tab'('	') y 'new line'('
') no se tiene acción nula, mientras que para cualquier otro carácter no existe acción especificada y por tanto se ejecuta la acción por default, que es la impresión en pantalla.

/* Programa: Ej1.1 */

%%

[ [\fn] ];

UNIX: Una puerta abierta a los sistemas abiertos
b) Este programa en lex realiza un conteo de caracteres, palabras y número de líneas de un texto. Esto se logra mediante la identificación de \n lo cual nos indicará fin de línea y por tanto el contador de líneas se incrementará; la identificación de cualquier cosa menos 'blanco', \t o \n con lo que identificaremos una palabra y el número de caracteres del mismo; y finalmente con la identificación de cualquier otro carácter menos \t con lo que reconoceremos caracteres adicionales.

/* Programa: Ej2.1 */

```
int nchar, nword, nline;


++nchar;
++nline;

[^ \n\t] {
    ++nword;
    nchar+=yyleng;
}

{  
    ++nchar;
}

main()
{
    yylex();
    printf("Caracteres=%d\n Palabras=%d\n Lineas=%d\n", nchar, nword, nline);
}
```

c) Este programa en Lex permite la impresión de las líneas analizadas, precedidas del número de línea. Esto se logra mediante la utilización de la variable yylineno la cual guarda el número de línea que se está analizando.

/* Programa: Ej3.1 */
Este programa en lex nos permite llevar la cuenta del número de palabras de una misma longitud, es decir por ejemplo:

- 5 palabras de longitud 3.
- 3 palabras de longitud 5, etc.

Esto se consigue mediante la utilización de arreglos, en donde el índice de este arreglo corresponderá a la longitud de cada palabra.

/* Programa: Ej4.1 */

```c
int longitud[100];

YVW
Y

for (i=4, i<100, i++)
    if longitud[i]>0
        printf("%d	%d\n", i, longitud[i]);

return(1);
}
```

d) Este programa en Yacc nos permitirá la generación de una gramática para calculadora. Recibirá los símbolos de un programa en lex y chequeará la consistencia del orden de las palabras para que pueda funcionar como una calculadora.

/* Programa: Ej1.y */

```c
%token _constante
%token _mas
```

UNIX: Una puerta abierta a los sistemas abiertos
%%
linea: expresion
{ printf("RESULTADO=%fn",$1);
};

expresion: expresion _mas expresion
 {$$=$1+$3;}
 /
expresion _menos expresion
 {$$=$1-$3;}
 /
expresion _por expresion
 {$$=$1*$3;}
 /
expresion _div expresion
 {$$=$1/$3;}
 /
_constante
 {$$=$1;};

UNIX: Una puerta abierta a los sistemas abiertos
Capítulo 5
LLAMADAS AL SISTEMA UNIX

5.1 ARQUITECTURA DEL SO UNIX

El sistema operativo UNIX presenta la arquitectura que se esquematiza a continuación:

![Diagrama de arquitectura de UNIX]
5.1.1 **SUBSISTEMA DE FICHEROS**

Maneja ficheros, administra espacios libres a los ficheros, dá espacio a los ficheros, permite recuperar datos y dá el acceso a ficheros.

5.1.2 **SUBSISTEMA DE DISPOSITIVOS**

a) Dispositivo de bloque: Trabaja con bloques enteros. Un ejemplo de este tipo de dispositivos es el disco.

b) Dispositivo de carácter: Dá la impresión de trabajar de carácter por carácter. Un ejemplo de este tipo de dispositivos es la terminal.

c) Manejadores de dispositivos: Es un programa que permite la interface del hardware, es decir, permiten la manipulación de los diferentes dispositivos como teclado, impresora, etc.

5.1.3 **SUBSISTEMA DE CONTROL DE PROCESOS**

a) Se refiere a la comunicación entre procesos. (generalmente son jerárquicos.

b) Calendarizador: calcula prioridades, quantum, permite la sincronización entre procesos.

c) Administrador de memoria: determina que procesos van a estar en memoria para ser ejecutados. Se relaciona directamente con el subsistema de ficheros.

5.1.4 **CONTROL DE HARDWARE**

El control de hardware está escrito en lenguaje ensamblador, y es aquel que controla toda la parte hardware existente del equipo en el cual se encuentra funcionando el sistema operativo UNIX.

5.2 **LLAMADAS AL SISTEMA**

Las llamadas del sistema son el nivel más bajo de interacción con el sistema operativo UNIX. Estas son las entradas al núcleo. Son las funciones que ofrecen el sistema operativo; todo lo demás se construye sobre ellas.

Para la comprensión de esta parte, es impresindible presentar una breve revisión de la estructura del sistema de ficheros.

Un directorio es un fichero que contiene una lista de nombres de otros ficheros y el lugar donde se encuentran. El "lugar" es en realidad un índice (único para cada fichero) en otra tabla llamada tabla de i-nodos. El i-nodo es el lugar donde se guarda toda la información sobre un fichero, excepto su nombre. Por lo tanto, una entrada de directorio consta únicamente de dos cosas, el número del i-nodo y el nombre del fichero.

La información que posee un i-nodo es la siguiente:
Propietario del fichero.

Permisos de acceso.

Tiempo del último acceso.

Indicación de dónde está el fichero.

Los i-nodos se almacenan en el sistema de ficheros pero si el Kernell se inicializa, este se carga a memoria principal.

<table>
<thead>
<tr>
<th>bloque</th>
<th>super</th>
<th>lista</th>
<th>bloques</th>
</tr>
</thead>
<tbody>
<tr>
<td>boot</td>
<td>block</td>
<td>i-nodos</td>
<td>datos</td>
</tr>
</tbody>
</table>

Tanto el bloque boot como el super block necesariamente existen, y su tamaño es fijo según la versión.

- El bloque boot, contiene el código para hacer el boot (carga parámetros para el sistema operativo).
- El Super block, describe el estado de sistemas de ficheros. (cuán largo es, dónde encontrar espacio libre, etc.).
- La lista de i-nodos, es fijada por el administrador del SO. Los i-nodos referencian a los ficheros por medio de índices. El nodo que necesariamente debe existir es el del root.

Toda la entrada y salida se efectúa leyendo o escribiendo ficheros, ya que todos los dispositivos periféricos, incluyendo la terminal, son ficheros en el sistema de ficheros. Esto significa que una sola interfaz maneja todas las comunicaciones entre el programa y los dispositivos periféricos.

En el caso más general, antes de leer o escribir un fichero, es necesario comunicar al sistema tal intento; este proceso se conoce como abrir el fichero. Si se va a escribir en un fichero, puede ser necesario crearlo. El sistema revisa el permiso para hacerlo, y si todo va bien regresa un entero no negativo llamado descriptor de fichero. Siempre que se haga E/S en el fichero, se utiliza el descriptor del alchivo para identificarlo y no el nombre. El sistema mantiene toda la información sobre un fichero abierto, un programa se refiere a dicho fichero únicamente por su descriptor de fichero.

Hay arreglos especiales para facilitar la entrada y salida por medio de la terminal. Cuando es iniciado por el shell, un programa adquiere tres ficheros abiertos con descriptores de fichero 0, 1 y 2 conocidos como entrada estándar, salida estándar y error estándar. Todos están conectados a la terminal; así que si un programa únicamente lee el descriptor de fichero 0 y escribe en los descriptores de fichero 1 y 2, puede hacer E/S sin tener que abrir ficheros. Si abre algún otro fichero, éstos tendrían descriptores de fichero 3, 4, etc.
Si la E/S se redirecciona hacia ficheros o interconexiones o desde ellos, el shell cambia las asignaciones de los descriptores de archivo 0 y 1 de la terminal a los ficheros mencionados. Normalmente el descriptor de fichero 2 permanece asignado a la terminal para que reciba los mensajes de error.

a) Tabla de i-nodos: Es una tabla para todo el sistema, es dinámica y posee un contador de cuantas referencias hay para un fichero determinado.

b) Tabla de ficheros general: Es también una tabla para todo el sistema, y guarda información de como ha sido abierto el fichero. Hace referencia a la tabla anterior. Aquí se incluye un apuntador al último fichero referenciado para lectura/escritura.

c) Tabla de ficheros del usuario: Existe una para cada usuario y contiene a los escritores de fichero.

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>c</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1 a read</td>
<td>2</td>
</tr>
<tr>
<td>1</td>
<td>1 b read</td>
<td>1 a</td>
</tr>
<tr>
<td>2</td>
<td>1 s write</td>
<td>1 b</td>
</tr>
<tr>
<td>3</td>
<td>L/E</td>
<td>1 s</td>
</tr>
<tr>
<td>4</td>
<td>1 k read</td>
<td>3 k</td>
</tr>
<tr>
<td>5</td>
<td>1 k R/W</td>
<td></td>
</tr>
</tbody>
</table>

5.2.1 DESCRIPCION DE ALGUNAS LLAMADAS AL SISTEMA

Las llamadas al sistema más utilizadas en el sistema Unix (OPEN, CLOSE, READ, WRITE, CREATE, PIPE, DUP, CHOWN, CHMODE) se presentan a continuación:

5.2.1.1 OPEN

Este comando nos sirve para la apertura de un fichero, tanto para para lectura como para escritura.

SINOPSIS

```c
#include <fcntl.h>

int open (path, oflag [, mode ])
char *path;
int oflag, mode;
```

DESCRIPCION
Open abre a un descriptor de fichero para el fichero nombrado y fija las banderas de estatus del fichero de acuerdo con el valor de oflag.

En caso de éxito, un número entero no negativo, el descriptor de fichero, es retornado.

Open fija el apuntador de fichero usado para marcar la posición presente dentro del inicio del fichero.

Open fija el nuevo descriptor de fichero para recordar llamadas al sistema a través de aperturas exec.

Notese que ningún proceso puede tener más de 64 descriptores de fichero abiertos simultáneamente.

**ARGUMENTOS**

Path apunta a un path de un fichero nombrado.

Los valores de Oflag son construidos por banderas de la siguiente lista:

Unicamente una de las siguientes tres banderas pueden ser usadas:

- **O_RDONLY** Apertura para lectura únicamente.
- **O_WRONLY** Apertura para escritura únicamente.
- **O_RDWR** Apertura para lectura y escritura.

Cualquier número de las siguientes banderas pueden ser colocados dentro de oflag.

- **O_NDELAY** Esta bandera puede afectar posteriores lecturas o escrituras.

  Cuando se realiza una apertura FIFO,
  - Si **O_NDELAY** es fijada y **O_RDONLY** es fijada, open retorna sin demora.
  - Si **O_NDELAY** es fijada y **O_WRONLY** es fijada, open retorna un error si no se ha dado un proceso actual de apertura de fichero para lectura.
  - Si **O_NDELAY** es definido y **O_RDONLY** es fijado, bloquea a open hasta que se dé un proceso de apertura de fichero para escritura.
  - Si **O_NDELAY** es definido y **O_WRONLY** es fijado, open es bloqueado hasta que se dé un proceso de apertura de fichero para lectura.

Cuando la apertura de una fichero es asociada con una línea de comunicación:

- Si **O_NDELAY** es fijado open retorna sin espera de acarreo.
- Si **O_NDELAY** es definido open es bloqueado hasta que se presente el acarreo.
- **O_NDELAY** no tiene efecto cuando la apertura de fichero está asociada con una línea de impresión paralela.

  - **O_APPEND** Si es fijado, open fija el puntero del fichero al final del fichero previo de cada escritura.

  - **O_CREAT** Si el fichero existe, esta bandera no tiene efecto. Caso contrario open creará el fichero y hace lo siguiente:
    - El ID del dueño del fichero es fijado a un efectivo ID de usuario del proceso.
    - El ID del grupo del fichero es fijado a un efectivo ID de grupo del proceso.
    - Los 12 bits de modo del fichero son fijados a un valor de modo con las siguientes modificaciones:
      - define todo el set de bits en la oculta creación del modo del fichero.
      - define la imagen de texto grabada antes del bit de ejecución de el modo.

  - **O_TRUNC** Si el fichero existe, trunca su longitud a 0, y no cambia ni el modo ni el dueño.

  - **O_EXCL** Si **O_EXCL** y **O_CREAT** son fijadas, open falla si el fichero existe.

  - **O_SYNC** Si es fijada, todas las posteriores escrituras son sincronizadas.

  Típicamente, el control es retornado al programa que realiza una escritura tan pronto como los datos han sido copiados dentro del buffer por medio del sistema operativo. La escritura actual al disco o a cinta puede ser demorada por algunos segundos. La escritura sincrónica espera hasta que los datos y todo lo necesario de la información del i-node ha sido escrita a disco después de retornar el control al programa de escritura.

**CONDICIONES DE FALLA**

Open falla y no realiza la apertura del fichero nombrado si uno o más de lo siguiente es verdadero:

  - El nombre del path es más largo que el máximo número de caracteres permisibles para el path. [ENOENT]
  - **O_CREAT** es fijado, el fichero no existe, y el directorio que contiene el fichero no permite la escritura. [EACCES]
  - Un componente del prefijo del path no es un directorio. [ENOTDIR]
  - **O_CREAT** no se ha fijado y el fichero nombrado no existe. [ENOENT]
  - Un componente del prefijo del path deniega permiso de búsqueda. [EACCES]
  - El permiso de Oflag es denegado para el fichero nombrado. [EACCES]
El fichero nombrado es un directorio y oflag es de escritura o lectura/escritura. [EISDIR]
El fichero nombrado reside en un fichero del sistema únicamente de lectura y oflag es de escritura o lectura/escritura. [EROFS]
El máximo número de descriptores de fichero (NOFILE) ha sido actualmente abierto. [EMFILE]
El fichero nombrado es un carácter especial o bloque especial de fichero, y el dispositivo asociado con este fichero especial no existe. [ENXIO]
El fichero es un procedimiento puro (texto compartido) que ha sido ejecutado y oflag es de escritura o lectura/escritura. [ETXTBSY]
El directorio que contiene el fichero no puede ser extendido, el fichero no existe y O_CREAT ha sido especificado. [ENOSPC]
El Path apunta fuera del espacio alocado de dirección del proceso. [EFAULT]
O_CREAT y O_EXCL son fijadas, y el fichero nombrado existe. [EXIST]
O_NDELAY es fijada, y el fichero nombrado es un FIFO, O_WRONLY es fijada, y ningún proceso tiene el fichero abierto para lectura. [ENXIO]
Una señal ha sido dada durante la llamada al sistema open. [EINVAL]
La tabla de ficheros del sistema está llena. [EINVAL]
El fichero existe, lockf(2), lockf(2X) o fcntl(2) se han usado para asegurar una porción del fichero, y un modo de ejecución mandatorio está en efecto en el fichero. [EAGAIN]

VALOR DE RETORNO

entero no negativo : terminación exitosa; el entero no negativo es el descriptor del fichero.
-1 : error; errno indica error

5.2.1.2 CLOSE
Este comando, nos permite cerrar un descriptor de fichero que ha sido abierto.

SINOPSIS

int close (fildes)
int fildes;

DESCRIPCION

UNIX: Una puerta abierta a los sistemas abiertos
fildes es un descriptor de fichero obtenido de una llamada al sistema como: creat, open, dup, fcntl, o pipe. Close cierra el descriptor de fichero indicado por fildes. Todas las marcas de los registros asegurados son removidos.

**CONDICIONES DE FALLO**

El comando Close fallará si fildes no es un válido o si el descriptor de fichero no está abierto. [EBADF]

**VALOR DE RETORNO**

- \( v \) 0 : terminó exitosamente
- \( v \) 1 : error; errno indica el error

### 5.2.1.3 READ

Este comando permite la lectura de un fichero que ha sido previamente abierto.

**SINOPSIS**

\[
\text{int read (fildes, buf, nbyte)}
\]

\[
\begin{align*}
\text{int fildes;} \\
\text{char *buf;} \\
\text{unsigned nbyte;}
\end{align*}
\]

**DESCRIPCION**

Read intenta la lectura de nbyte bytes de el fichero asociado con fildes dentro del buffer apuntado por buf. Fildes es el descriptor de fichero obtenido por el uso de una llamada al sistema como: creat, open, dup, fcntl, o pipe.

En dispositivos capaces de búsquedas, el read inicia en una posición dentro del fichero indicado por un puntero de fichero asociado con filde. El comando read incrementa el puntero de fichero de acuerdo al número de bytes leídos después de un read.

En dispositivos incapaces de búsquedas, siempre la lectura se hace de la posición actual. El valor del puntero de fichero asociado con cada fichero es indeterminado.

Luego de una terminación exitosa, read retorna el número de bytes actualmente leídos y los coloca en el buffer; este número puede ser menor que nbytes si el fichero es asociado con una línea de comunicación, o si el número de bytes izquierdos es menor que nbytes bytes.

Read retorna el valor de 0 cuando el fin-de-fichero a sido alcanzado.

Cuando se realiza una lectura de un direccionamiento vacío (o FIFO):

- \( v \) Si O_NDELAY es fijado, la lectura retorna un 0.
Si O_NONDELAY es borrado, la lectura es bloqueada hasta que los datos sean escritos al fichero, o el fichero no es abierto longitudinalmente para escritura.

Cuando se realiza la lectura de un fichero asociado con un tty los datos corrientes no podrán ser abilitados

Si O_NONDELAY es fijado, la lectura retorna un valor de 0.

Si O_NONDELAY es borrado, la lectura es bloqueada hasta que los datos estén disponibles.

Cuando se realiza la lectura de un fichero regular con una marca mandatoria de fichero/registro fijado, y este es bloqueado el seguro de escritura del segmento del fichero será leído.

Si O_NONDELAY es fijado, la lectura retornará -1 y se fijará errno en EAGAIN.

Si O_NONDELAY es borrado, la lectura se pausará hasta que el seguro del registro bloqueado sea removido.

**CONDICIONES DE FALLA**

Read falla si uno o más de los siguientes es verdadero:

- `Fildes` es un descriptor invalido de fichero abierto para lectura. [EBADF]
- `Buf` apunta fuera del espacio asignado para direccionesds. [EFAULT]
- Una señal ha sido capturada durante la llamada al sistema con read. [EINTR]
- El dispositivo asociado con el descriptor de fichero es una fichero especial de bloque o carácter especial, y el puntero de fichero está fuera de rango. [ENXIO]

**VALOR DE RETORNO**

- no-negativo : terminación exitosa;
- integer : número que indica el número de bytes actualmente leídos;
- 1 : error; errno indica el error

**5.2.1.4 WRITE**

Este comando permite la escritura en un fichero que ha sido previamente abierto.

**SINOPSIS**

```c
int write(fildes,buf,nbyte)
int fildes;
char *buf;
```
unsigned nbyte;

**DESCRIPCION**

Si el fichero que será escrito es un pipe (o FIFO) y la bandera O_NDELAY del fichero es fijada, entonces la escritura llenar un pipe (o FIFO) retorna una cuenta de 0. En otro caso (O_NDELAY definido), la escritura para llenar un pipe (o FIFO) será bloqueada hasta que exista espacio disponible.

Una escritura a un fichero regular es bloqueada si la región del fichero es bloqueada por otro proceso que ha usado lockf(2) o lockf(3X) o fcntl(2) con un seguro de fichero/registro mandatorio habilitado en el fichero. En estos casos, si lockf(2) fue usado para asegurar una región del fichero o si lockf fue usado con la bandera O_NDELAY fijada, write espera hasta que el seguro del registro bloqueado ha sido removido.

**CONDICIONES DE FALLA**

Write falla y no cambia el puntero del fichero si una o más de las siguientes condiciones e verádadera.

- `v` Fildes no es un descriptor abierto de fichero válido para escritura. [EBADF]
- `v` Un intento es hecho para escribir a un pipe que no ha sido abierto para lectura por cualquier proceso [EPIPE y la señal SIGPIPE]
- `v` Un intento es hecho para escribir a un fichero que excede el límite de tamaño de fichero de el proceso o el máximo tamaño de fichero. [EFBIG]
- `v` Cuando no existe espacio libre disponible en el dispositivo que contiene al fichero. [ENOSPC]
- `v` Un error físico de I/O ha ocurrido. [EIO]
- `v` El dispositivo asociado con el descriptor de fichero es un bloque especial o un carácter especial de fichero y el valor del puntero del fichero está fuera de rango. [ENXIO]
- `v` Buf apunta fuera del espacio de direcciones de alocación del proceso. [EFAULT]
- `v` Una señal fue dada durante la llamada al sistema write. [EINVAL]
- `v` Un seguro de fichero/registro mandatorio está en efecto, O_NDELAY ha sido fijado, y existe un seguro de registro bloqueado. [EAGAIN]
- `v` La cantidad total de memoria disponible del sistema cuando una lectura vía I/O es temporalmente insuficiente. [EAGAIN]
- `v` La tabla de seguridad de registros del sistema está llena, por lo que write no puede esperar hasta que la seguridad de registros sea removida. [ENOLCK]
Cuando el write está esperando y causa un deadlock debido a que un seguro mandatorio previo en el fichero. [EDEADLK]

**VALOR DE RETORNO**

- entero no negativo: terminación exitosa; el entero no negativo indica el actual número de bytes escritos.
- 1: error; errno indica error

### 5.2.1.5 CREATE

Este comando permite la creación de un nuevo fichero o la sobreescritura de un fichero ya existente.

#### SINOPSIS

`int creat (path, mode)`

`char *path;`

`int mode;`

#### DESCRIPCIÓN

`Creat` permite crear un fichero ordinario nuevo o preparar para la sobreescribir uno de un fichero existente nombrado por el nombre (path) y apuntado por el identificador path.

Si el fichero existe, `create` lo truncará a una longitud de 0, sin cambiar ni el modo ni el propietario. El comando `creat` fija el identificador (ID) del propietario del fichero y fija el identificador de grupo (ID) para los procesos, y los 12 bits de modo del fichero son fijados a un valor de mode, con las siguientes modificaciones:

- Creat borra todos los bits fijados en el modo de creación de fichero recubierto por el proceso.
- Creat borra el bit en el modo de grabación de texto después de la ejecución.

Luego de la terminación exitosa, `create` retorna un valor entero no-negativo, que representa el descriptor de fichero; el fichero es abierto para escritura, aun si el modo no permite la escritura. El puntero de fichero apunta al inicio del fichero. El descriptor de fichero es fijado para permanecer abierto para cualquier llamada al sistema.

No se podrán manipular más de 64 ficheros abiertos simultáneamente.

#### CONDICIONES DE FALLO

- Creat falla si uno o más de los siguientes es verdadero:
  - Una componente del prefijo del path no es un directorio. [ENOTDIR]
  - Una componente del prefijo del path no existe [ENOENT]
  - Permiso de búsqueda denegada a una componente del prefijo del path. [EACCES]

**UNIX:** Una puerta abierta a los sistemas abiertos
El nombre del path es nulo. [ENOENT]
El fichero no existe y el directorio en donde se desea crear el fichero no permite la escritura del mismo. [EACCES]
El nombre del fichero recibe en el sistema de ficheros de solo-lectura. [EROFS]
El fichero existe y el permiso de escritura es denegado. [EACCES]
El nombre del fichero es igual al de un directorio existente. [EISDIR]
El máximo número de usuarios para el descriptor de ficheros está tomado. [EMFILE]
El path apunta fuera del espacio reservado de direcciones para el proceso. [EFAULT]
La tabla del sistema de ficheros está llena. [ENFILE]

**VALOR DE RETORNO**
Con una terminación exitosa, un entero no-negativo será retornado, el cual corresponde al descriptor de fichero. De otra forma un valor -1 es retornado, y el errno es fijado para indicar el error.

### 5.2.1.6 PIPE
Este comando permite la creación de canales inter-procesos.

**SINOPSIS**

```
int pipe(fildes)
int fildes[2];
```

**DESCRIPCIÓN**

Los buffers del pipe escriben sobre los 5120 bytes de datos antes de bloquear el proceso de escritura.

En los procesos no se podrá tener más de 64 descriptores de ficheros abiertos simultáneamente.

**CONDICIONES DE FALLO**

- El comando pipe falla si más de 63 descriptores de ficheros están abiertos al mismo tiempo. [EMFILE]
- La tabla del sistema de ficheros está llena. [ENFILE]

**VALORES DE RETORNO**

- 0 : terminación exitosa
- 1 : error; errno inindica el error
5.2.1.7 DUP

El comando dup, duplica el descriptor de una fichero abierto.

**SINOPSIS**

```c
int dup (fildes)
int fildes;
```

**DESCRIPCIÓN**

El comando dup retorna un descriptor de ficheros teniendo común con fildes, teniendo en común el descriptor de ficheros obtenido en las llamadas al sistema creat, open, dup, fcntl, o pipe:

- El mismo fichero abierto (o pipe).
- El mismo puntero de fichero. (i.e., ambos descriptores de ficheros comparten un puntero de fichero).
- El mismo modo de acceso (read, write or read/write).
- Dup fija el nuevo descriptor de fichero para permanecer abierto por medio de la llamada al sistema exec.
- Dup retorna el descriptor más bajo de fichero disponible.

**CONDICIONES DE FALLO**

El comando dup fallará si uno a más de los siguientes puntos es verdadero:

- `v Fildes no es un descriptor de fichero abierto válido. [EBADF]`
- `v 64 descriptores de fichero están abiertos simultáneamente. [EMFILE]`

**VALOR DE RETORNO**

- `v no-negativo : terminación exitosa`
- `v entero : es el descriptor de fichero`
- `v 1 : error; errno indicativo de error`

5.2.1.8 CHOWN

El comando chown permite cambiar el dueño y grupo de un fichero específico o un grupo de óllos.

**SINOPSIS**

```c
int chown (path, owner, group)
char *path;
int owner, group;
```

**DESCRIPCIÓN**

Chown fija el ID del dueño y el ID del grupo del fichero nombrado con los valores contenidos en el dueño y grupo respectivamente. El path apunta al path name nombrando al fichero.

UNIX: Una puerta abierta a los sistemas abiertos
Solo procesos con un ID de usuario efectivo igual al dueño del fichero o un super usuario pueden cambiar la propiedad de un fichero.

Si un usuario distinto al super-usUARIO invoca chown, chown define los bits del set-user-ID y del set group ID bit del modo del fichero, 04000 y 02000 respectivamente.

**CONDICIONES DE FALLA**

Chown falla y no cambia el dueño ni el grupo del fichero nombrado si uno o más de las siguientes condiciones se da:

- Un componente del prefijo del path no es un directorio. [ENOTDIR]
- El fichero nombrado no existe. [ENOENT]
- El argumento del path name es demasiado largo. [ENOENT]
- El permiso de búsqueda es denegado sobre un componente del prefijo del path. [EACCES]
- El ID del usuario efectivo no corresponde al dueño del fichero y el ID del usuario no es un super-usuario. [EPERM]
- El fichero nombrado reside en un fichero del sistema con permiso sólo para lectura. [EROFS]
- El Path apunta fuera del espacio de dirección de alocación del proceso. [EFAULT]

**VALOR DE RETORNO**

- 0 : terminación exitosa
- 1 : error; errno indica error

**5.2.1.9 CHMODE**

El siguiente comando nos permite el cambio de modo de los ficheros.

**SINOPSIS**

```c
int chmod (path, mode)
char *path;
int mode;
```

**DESCRIPCIÓN**

Chmod fija una porción del modo de los permisos de acceso para un fichero especificado por el path de acuerdo a un patrón de bits contenidos en mode. Path apunta al path y nombre del fichero.

Los bits de permiso de acceso se interpretan como siguen:

- 04000  Fija a un ID de usuario para ejecución.
- 02000  Fija a un ID de grupo para ejecución.
- 01000  Salva una imagen del texto después de la ejecución.
v 00400 Lectura para el propietario
v 00200 Escritura para el propietario
v 00100 Ejecución (o búsqueda de un directorio) para el propietario
v 00070 Lectura, escritura, ejecución (búsqueda) para el grupo
v 00007 Lectura, escritura, ejecución (búsqueda) para otros

Solo un proceso con un ID de usuario efectivo igual al del dueño del fichero o un super usuario puede cambiar el modo del fichero para el propietario, para el grupo o para otros usuarios.

CONDICIONES DE FALLA

El comando chmod fallará y no se podrá realizar el cambio de modo si una o más de las siguientes condiciones falla:

v Una componente del prefijo del path no es un directorio. [ENOTDIR]
v El nombre del fichero no existe. [ENOENT]
v El permiso de búsqueda es denegado para una componente del prefijo del path. [EACCES]
v El ID de usuario efectivo no corresponde al dueño del fichero o al de un super-usuario. [EPERM]
v El nombre de fichero reside en un sistema de ficheros de sólo-lectura. [EROFS]
v El path apunta afuera de la localización de espacio de direcciones. [EFAULT]

VALOR DE RETORNO

v 0 : terminación exitosa
v 1 : error; errno indicativo de error

5.2.2 EJERCICIO PRACTICO

El marco teórico antes explicado nos introdujo en el uso de determinadas llamadas al sistema, así por ejemplo: create, open, read, and write, a partir de los cuales presentamos la implementación de un programa para la manipulación de ficheros.

5.2.2.1 Listado del Programa

```
#include <fcntl.h>
#include <stdio.h>

int fd1, fd2, fd3;
main (argc, argv)
    int argc;
```

UNIX: Una puerta abierta a los sistemas abiertos.
char *argv[ ];
char c[511];
{
    if (argc != 4) exit(-1)
    if (fd1 = creat(argv[1], 0666)) == -1)
    {
        printf("No puedo crear \%s\n", argv[1]);
        exit(-1)
    }
    if (fd2 = open(argv[2], O_RDONLY)) == -1)
    {
        printf("No puedo abrir \%s\n", argv[2]);
        exit(-1)
    }
    if (fd3 = open(argv[3], O_RDONLY)) == -1)
    {
        printf("No puedo abrir \%s\n", argv[3]);
        exit(-1)
    }
    printf("Descriptor fd1=%d; fd2=%d; fd3=%d\n", fd1,fd2,fd3)
    if (read (fd2, c, sizeof(c)) != 1 )
        write(fd1, c, sizeof(c));
    if (read (fd3, c, sizeof(c)) != 1 )
        write(fd1, c, sizeof(c));
    if (read (fd2, c, sizeof(c)) != 1 )
        write(fd1, c, sizeof(c));
}

5.2.2.2 Descripción del programa
En este programa se incluyen las librerías: stdio.h en la cual está definida la macro fileno (fp) que devuelve el descriptor de un fichero y fcntl.h para el uso de las banderas utilizadas en las llamadas al sistema.
Las variables: fd1, fd2, y fd3 son los descriptores de los ficheros.
El programa principal main recibe argc argumentos en el arreglo argv. El número de argumentos para la ejecución de este programa no deberá ser inferior a 4, los cuales serán: a) el nombre del programa, b) el nombre del fichero de salida y c,d) los nombres de los ficheros de datos (entrada).

Al inicio del programa se crea el fichero de salida (argv[1]) y se le da permisos de escritura y lectura para todos (0666), en caso de ser válida la creación se procede a abrir los ficheros de datos (argv[2] y argv[3]) en modo de lectura (O_RDONLY).

Si no existe ningún tipo de error en la apertura de los ficheros, se imprime por pantalla los valores de los descriptores fd1, fd2 y fd3, luego de lo cual se emplea la función read para accesar al fichero del descriptor fd2 y leer de él un bloque de 512 bytes y almacenarlos en el buffer c, este buffer luego es escrito en el fichero del descriptor fd1; seguidamente repetimos este proceso, pero, ahora teniendo como fichero fuente al del descriptor fd3 y como destino a fd1, y finalmente realizamos el mismo proceso teniendo como fuente a fd2 y como destino a fd1.

Este programa fue probado de las siguientes formas:

- Mandándole de parámetros un fichero A (vacío) para la salida, y dos ficheros C y D con el mismo contenido como datos.
- Mandándole de parámetros un fichero A (no vacío) para la salida, y dos ficheros C y D con el mismo contenido como datos.
- Mandándole de parámetros un fichero A para la salida, y dos ficheros C y D con distinto contenido como datos.
- Mandándole de parámetros un fichero A para la salida, y el mismo fichero C en las dos entradas como datos.
Capítulo 6
ADMINISTRACION DEL SISTEMA

Unix requiere de la supervisión y control de su operación para garantizar el máximo rendimiento en su funcionamiento. Para optimizar el funcionamiento en un ambiente específico es conveniente “afinar” los parámetros del sistema.

Toda instalación Unix debe tener una persona a cargo de su mantenimiento y operación; el administrador del sistema, esta persona tiene rango de superusuario y requiere de privilegios especiales.

Todo sistema Unix tiene una cuenta de usuarios especial: root (superusuario). La mayoría de variantes de Unix tienen otra cuenta para el administrador del sistema: sa, sysadm, admin, etc.

Muchas variantes de Unix incorporan software específico para la administración del sistema (sistema de menús). Este software, sin embargo, accesa siempre a utilitarios propios del sistema operativo.

Los procedimientos de administración varían de acuerdo a las diferentes versiones de los sistemas (Unix V, Xenix, Centix, Ultrix, AIX, etc.) pero la filosofía básica es la misma.

A continuación se revisará de forma resumida las partes más importantes que deben tenerse en cuenta en la administración del sistema Unix:

- Iniciación y apagado del equipo.
- Administración de usuarios y seguridades.
- Administración del sistema de ficheros.
Contabilidad del sistema.
- Respaldos (backup) y recuperación (restore).
- Configuración de periféricos.
- Sistemas de comunicación.
- Control de procesos den el tiempo.

6.1 INICIACION Y APAGADO DEL EQUIPO

6.1.1 INICIACION DEL EQUIPO
Durante la iniciación del equipo, se deberá tener en cuenta los siguientes puntos:

- Revisar el sistema de ficheros raíz (si es necesario)
- Se selecciona el modo de operación (mono o multiusuario)
- Se montan los sistemas de ficheros secundarios (ver Administración del Sistema de Ficheros pto. 3).

La revisión del sistema raíz es necesaria, si el último apagado no fue normal. Esta revisión puede extenderse al sistema de ficheros secundarios.

Unix entra, por omisión (default), a modo monousuario. La terminación del procesos inicial fuerza la enmtrada a modo multiusuario.

La secuencia de eventos durante el prendido del equipo está consignada en el fichero /etc/inittab que es accedido por /etc/init que es el padre de todos los procesos del sistema.

6.1.2 APAGADO DEL EQUIPO
Durante el apagado del equipo, se deberá tener en cuenta los siguientes puntos:

- Se desmontan todos los sistemas de ficheros (ver Administración del Sistema de Ficheros pto. 3).
- Se sincronizan todos los bitfers de E/S del sistema.

La sincronización graba en disco la imagen en memoria de los sistemas de ficheros y efectúa todas las operaciones del disco pendientes.

El desmontaje del sistema de ficheros remueve la marca de "en uso" de los mismos, haciendo posible un montaje futuro normal.

Durante el procesos de apagado se suele avisar a los usuarios activos y congelar sus procesos después de un tiempo prudencial.
6.2 ADMINISTRACION DE USUARIOS

Para poder accesar a un sistema Unix es necesario especificar una identificación de usuario. En Unix cada usuario está adscrito a un grupo de usuarios. Normalmente los grupos corresponden a dependencias dentro de la organización. El password (contraseña) de cada usuario es opcional. Cada usuario tiene un directorio de trabajo y un shell (intérprete de comandos).

Los grupos se definen en el fichero /etc/group. Los usuarios se definen en el fichero /etc/passwd.

Estos son ficheros de texto normales y pueden ser modificados empleando un editor de textos convencional. Diferentes versiones de Unix ofrecen utilitarios para administrar los usuarios sin manipular directamente estos ficheros.

La única forma de definir passwords es mediante el comando passwd. Este comando guarda la contraseña en el fichero /etc/passwd en forma encriptada. El sistema de seguridad de Unix es robusto entanto se mantenga la privacidad de los passwords.

Los ficheros de Unix tienen permisos de lectura, escritura y ejecución a nivel de usuarios y a nivel de grupo.

6.2.1 EL FICHERO DE PASSWORDS

El fichero de passwords define las cuentas de usuarios en Unix. Cada línea de entrada al fichero /etc/passwd, define una cuenta. Cada línea consiste de siete campos, separados unos de otros mediante los "dos puntos" (:). En la figura a continuación se muestra un ejemplo de una entrada de una cuenta típica de usuario:

```
sgarces:8d0fOt.d2Bkus:100:26:Santiago Garces:/usr acct/sgarces:/bin/csh
```

| | | | | | | |
| | | | | | | |
| | | | | | | |
| | | | | | | |
| | | | | | | |
| | | | | | | |
| | | | | | | |
| | | | | | | |
| | | | | | | |
| | | | | | | |
| | | | | | | |
| | | | | | | |
| | | | | | | |
| | | | | | | |
| | | | | | | |
| | | | | | | |
| | | | | | | |
| | | | | | | |
| | | | | | | |
| | | | | | | |
| | | | | | | |
| | | | | | | |
| | | | | | | |
| | | | | | | |
| | | | | | | |
| | | | | | | |
| | | | | | | |
| | | | | | | |
| | | | | | | |
| | | | | | | |
| | | | | | | |
| | | | | | | |
| | | | | | | |
| | | | | | | |
| | | | | | | |
| | | | | | | |
| | | | | | | |
| | | | | | | |
| | | | | | | |
| | | | | | | |
| | | | | | | |
| | | | | | | |
| | | | | | | |
| | | | | | | |
| | | | | | | |
| | | | | | | |
| | | | | | | |
| | | | | | | |
| | | | | | | |
| | | | | | | |
| | | | | | | |
| | | | | | | |
| | | | | | | |
| | | | | | | |
| | | | | | | |
| | | | | | | |
| | | | | | | |
| | | | | | | |
| | | | | | | |
| | | | | | | |
| | | | | | | |
| | | | | | | |
| | | | | | | |
| | | | | | | |
| | | | | | | |
| | | | | | | |
| | | | | | | |
| | | | | | | |
| | | | | | | |
| | | | | | | |
| | | | | | | |
| | | | | | | |
| | | | | | | |
| | | | | | | |
| | | | | | | |
| | | | | | | |
| | | | | | | |
| | | | | | | |
| | | | | | | |
| | | | | | | |
| | | | | | | |
| | | | | | | |
| | | | | | | |
| | | | | | | |
| | | | | | | |
| | | | | | | |
| | | | | | | |
| | | | | | | |
| | | | | | | |
| | | | | | | |
| | | | | | | |
| | | | | | | |
| | | | | | | |
| | | | | | | |
| | | | | | | |
| | | | | | | |
| | | | | | | |
| | | | | | | |
| | | | | | | |
| | | | | | | |
| | | | | | | |
| | | | | | | |
| | | | | | | |
| | | | | | | |
| | | | | | | |
| | | | | | | |
| | | | | | | |
| | | | | | | |
| | | | | | | |
| | | | | | | |
| | | | | | | |
| | | | | | | |
| | | | | | | |
| | | | | | | |
| | | | | | | |
| | | | | | | |
| | | | | | | |
| | | | | | | |
| | | | | | | |
| | | | | | | |
| | | | | | | |
| | | | | | | |
| | | | | | | |
| | | | | | | |
| | | | | | | |
| | | | | | | |
| | | | | | | |
| | | | | | | |
full name = nombre completo (descripción)
home directory = directorio hogar del usuario
startup program = programa inicial de arranque

6.2.2 EL FICHERO DE GRUPOS

El fichero de grupos contiene los nombres de los grupos por omisión (default) y grupos adicionales definidos, los cuales son referenciados en el fichero de passwords. Cada línea de entrada al fichero /etc/group, define un grupo. Cada línea consiste de cuatro campos, separados unos de otros mediante los "dos puntos" (:) al igual que en el fichero de passwords. En la figura a continuación se muestra un ejemplo de una línea de entrada de un grupo típico:

```
users:100:egorooe,boooat
          |     |     | account names
          |     |     |
          |     | GID
          |     | password
          |     group name
```

Donde:

- group names = nombre del grupo
- password = password para el grupo
- GID = identificador de grupo
- account names = nombres de cuentas

6.3 ADMINISTRACION DEL SISTEMA DE FICHEROS

Los nuevos sistemas de ficheros (residentes en unidades de disco) se definen mediante del comando mkfs (make file system). Mediante este comando se definen el máximo número de bloques y el máximo número de inodos por sistema de ficheros. Un mismo disco puede estar dividido en varias particiones. En cada partición puede definirse un sistema de ficheros separado.

Para poder utilizar un sistema de ficheros, este debe "montarse" mediante el comando mount. mount hace accesible el sistema de ficheros ubicándolo bajo un directorio y marcándolo como "en uso".

Antes de apagar el sistema es necesario desmontar los sistemas de ficheros, mediante el comando umount. umount sincroniza el sistema de ficheros y remueve la marca de "en uso".

UNIX: Una puerta abierta a los sistemas abiertos
Si el sistema no fue apagado normalmente y, por tanto, no se removió la marca "en uso" con el comando unmount, es necesario revisar los sistemas de ficheros para verificar su consistencia y eventualmente repararlos. Esto se logra mediante el comando fsck que revisa y reporta sistemas de ficheros, removiendo la marca de "en uso". Para el sistema de ficheros raíz puede ser necesario el reiniciar el sistema de ejecución de un fsck que lo reparará.

6.4 CONTABILIDAD DEL SISTEMA

Unix posee primitivas para registrar la ejecución de procesos, sutiempo, usuario y otros datos. Basados en estas primitivas se ha desarrollado un módulo de accounting o contabilidad del uso del sistema. Esta facilidad permite reportar el uso de CPU y espacio en disco por usuario, grupo y terminal. Adicionalmente, permite calcular costos de uso del sistema para cobro de servicio de dependencias u organizaciones.

La facilidad contable puede ser activada o desactivada a voluntad, normalmente en el proceso de prendido del equipo. La contabilidad puede afectar el tiempo de respuesta del equipo y requiere de una cuidadosa planificación.

Puesto que esta facilidad utiliza ficheros temporales, es de suma importancia incluirlos en las operaciones diarias de backup y prever mecanismos de recuperación para posibles fallas del equipo.

6.5 RESPALDOS Y RECUPERACION

Existen dos categorías de "backup":

- Copia de grupos de ficheros aislados.
- Copia total del sistema de ficheros.

6.5.1 COPIA DE GRUPOS DE FICHEROS AISLADOS

La copia de ficheros aislados se practica sobre ficheros dinámicos que experimentan cambios diarios y deben respaldarse con frecuencia. Para estos casos, los comandos tar y cpio son adecuados, dada su capacidad de copiar selectivamente, a diskette o cinta, una lista de ficheros dados. Esta forma particular de backup puede ser practicada indistintamente por el administrador del sistema o por usuarios particulares.

6.5.2 COPIA TOTAL DEL SISTEMA DE FICHEROS

La copia total del sistema de ficheros debe ser ejecutada periódicamente por el administrador del sistema como una forma de restituir la totalidad del sistema después de un daño de grandes proporciones como el daño físico del disco o la corrupción total del sistema de ficheros. Para este efecto se emplea los comandos dump y restore que permiten respaldar y restituir sistemas de ficheros basados en fechas de modificación, razón por la cual se les denomina incrementales.
Algunas versiones de Unix ofrecen facilidades de copia de sistemas de ficheros que pueden emplearse para restituir los mismos como si se tratase de la instalación de la primera vez (ej: vcopy).

6.6 CONFIGURACION DE PERIFERICOS

Además de la conexión física de periféricos (terminales e impresoras) al sistema, es necesario indicarle a éste las características de operación de los dispositivos. Esto incluye la velocidad de comunicación, número de bits por byte, de parada, de paridad y otros parámetros. Las diferentes variantes de Unix emplean mecanismos, ficheros y formatos diversos para representar y controlar estas características (Unix: inittab; Xenix: telinit).

Los principales tipos de periféricos que podemos distinguir en este sistema son:

- Terminales locales
- Terminales remotas
- Impresoras serie
- Impresoras paralelas

Otras periféricos, tales como unidades de cinta y diskette, están preconfigurados en el kernel y no requieren configuración de parámetros. Este tipo de periféricos se manejan a través de "drivers" que forman parte del kernel.

Normalmente, nuevos dispositivos traen consigo el "software" de instalación que adiciona el "driver" correspondiente al sistema operativo.

6.7 CONTROL DE PROCESOS EN EL TIEMPO

Unix provee de facilidades para ejecutar procesos en el futuro de forma automática.

El comando cron inspecciona periódicamente una lista de procesos ejecutados aquellos para los cuales se cumpla la fecha y hora. Esta fecha y hora puede ser repetitiva (ej: todos los lunes a las 07h00). Un uso típico de esta facilidad es ejecutar automáticamente backups a altas horas de la noche.

Muchas tareas de administración pueden ejecutarse de forma automática sin requerir la intervención directa del operador o del administrador del sistema. Entre estas tareas se destacan:

- Backups masivos.
- Transmisión/recepción de ficheros por comunicación.
- Ejecución de procesos de contabilidad del sistema.

Algunos utilitarios UNix hacen uso de esta facilidad. calendar examina diariamente a las 13h00 los calendarios de actividades de los usuarios seleccionando aquellas aplicables a la fecha y enviándolas por correo.
BIBLIOGRAFÍA

◊ Manuales del Sistema Operativo UNIX V.
◊ Manuales del Sistema Operativo SunOS.
◊ THOMAS Rebecca y FARROW Rik, UNIX Administration Guide for System V, Prentice-Hall, New Jersey.