UNIVERSIDAD NACIONAL AUTONOMA DE MEXICO

FACULTAD DE ESTUDIOS SUPERIORES

"CUAUTITLAN"

PROGRAMACION DEL SHELL PARA EL SISTEMA OPERATIVO UNIX/XENIX

T E S I S

QUE PARA OBTENER EL TITULO DE:

INGENIERO MECANICO ELECTRICISTA

P R E S E N T A:

MARLON ENRIQUE CZERMAK ANDRADE

DIRECTOR DE TESIS:
ING. ROGELIO RAMOS CARRANZA

TESIS CON
FALLA DE ORIGEN

Cuautitlán Izcalli, Edo. de Méx. 1991

UNAM – Dirección General de Bibliotecas Tesis Digitales Restricciones de uso

DERECHOS RESERVADOS © PROHIBIDA SU REPRODUCCIÓN TOTAL O PARCIAL

Todo el material contenido en esta tesis está protegido por la Ley Federal del Derecho de Autor (LFDA) de los Estados Unidos Mexicanos (México).

El uso de imágenes, fragmentos de videos, y demás material que sea objeto de protección de los derechos de autor, será exclusivamente para fines educativos e informativos y deberá citar la fuente donde la obtuvo mencionando el autor o autores. Cualquier uso distinto como el lucro, reproducción, edición o modificación, será perseguido y sancionado por el respectivo titular de los Derechos de Autor.

I N D I C E

PROGRAMACION DEL SHELL PARA EL SISTEMA OPERATIVO UNIX/XENIX

INTRODUCCION 4

CAPITULO 1 BREVE RESEÑA DEL SISTEMA OPERATIVO UNIX 8

Antecedentes 9
Núcleo del sistema operativo 13
Manejo de memoria 15
Manejo del procesador 16
Manejo de Entradas y Salidas 18
Manejo de archivos y de información 19
Lenguaje de control del sistema operativo 21
Comentarios finales 24

CAPITULO 2 CONCEPTOS BASICOS 26

Antecedentes 27
Archivos 27
archivos ordinarios 27
directorios de archivos 27
archivos especiales de dispositivos 28
estructura de los directorios 28
Organización de de los archivos del sistema 28
Convenciones 29
nombres de archivos 29
nombres de rutas 30
caracteres especiales 31
Comandos 32
línea de comandos 32
sintáxis 33
Entradas/Salidas 33
redireccionamiento 33
pipes (conductos) 34

CAPITULO 3 EL SHELL 35

Antecedentes 36
Conceptos básicos 36
comandos 36
como el shell busca los comandos 37
usando caracteres especiales en la linea de comandos 37
Mecanismos de las comillas 38
Redireccionando la entrada y/o salida 39
entrada y salida estandar 40
diagnóstico y otras salidas 40
línea de comandos y pipes (conductos) 41
comandos de substitución 43

Variables del shell 43
parámetros posicionales 44
variables definidos por el usuario 44
variables especiales predefinidas 47
Estado del shell 48
cambiando de directorio 48
el archivo .profile 49
Invocando al shell 49
Pasando argumentos o procedimientos del shell 49
Estructuras de control del shell 51
uso de la sentencia **if** 53
uso de la sentencia **case** 54
ciclos condicionales: **while** y **until** 55
ciclos para lista: **for** 55
control de ciclos: **breack** y **continue** 57
término de un procedimiento del shell : fin de archivo y salida ... 57
agrupacion de comandos: paréntesis y corchetes 58
redireccionamiento y entrada/salida y control de comandos . 59
transferencia a otros archivos y regreso:el comando . punto 59
Comandos de soporte y características 60
evaluación de condiciones: **test** 60
uso del comando **echo** 62
evaluación de expresiones: **expr** 63
salida del status: **true** y **false** 65
entrada de documentos en linea 65
redirección de entrada/salida usando descriptores 66
substitución condicional 67
invocación de banderas 68
Programacion efectiva y eficiente del shell 69
número de procesos generados 69
número de bytes de datos generados 71
abreviando la busqueda de directorios 71
busqueda ordenada de los directorios y la variable PATH ... 72

CAPITULO 4 EXPRESIONES REGULARES 73

Antecedentes 74
Expresiones regulares, Caracteres, Delimitadores, Cadenas sencillas ... 74
Caracteres especiales 76
Signos ^ y $ 78
Marcado de carácteres especiales 78
La equivalencia mas larga posible 79
Una expresion regular no excluye a la otra 79
Expresiones regulares vacias 80
Expresiones entre corchetes 80
La cadena de reposicion 81
Signo $ 81
Digitos marcados 82

CAPITULO 5 SINTAXIS DE ALGUNOS COMANDOS DEL SISTEMA OPERATIVO UNIX/XENIX 83

sed 84
awk 88
bc 92
du 96

CAPITULO 6 PROGRAMACION DE COMANDOS CON EL LENGUAJE SHELL 97

Programa copyto 98
Programa draft 98
Programa edfind 98
Programa edlast 99
Programa fsplit 99
Programa mkfiles 100
Programa null 100
Programa phone 101
Programa textfile 101
Programa writemail 102
Programa exe 102
Programa s80.c 103
Programa more 103
Programa load.sh 104
Programa vo 105
Programa TAR 107
Programa DF 116
Programa trad 117
Programa DU 119
Programa LPR 121
Programa off 123

GLOSARIO 126

BIBLIOGRAFIA 137

INTRODUCCION

Ultimante el sistema operativo UNIX/XENIX, ha tomado gran fuerza en el ámbito computacional, por lo que el desconocimiento de este sistema operativo es frecuente por parte de algunos usuarios que lo utilizan, esto es debido a la gama de comandos que éste sistema operativo posee (de 600 a 800 comandos aproximadamente, dependiendo de la versión del sistema operativo).

Uno de los comandos más importantes del sistema operativo es el comando **shell** (sh), que es un interprete entre el usuario y la computadora. El **Shell**, funciona de dos manera diferentes:

Como interprete de mandatos.

Procesa los mandatos introducidos en respuesta a su indicación.

Como lenguaje de programación de alto nivel

Se utiliza en la combinación de programas de utilidad estandar para crear aplicaciones complejas en minutos, en vez de dias. Es decir pueden usarse los propios comandos del sistema como subrutinas en los programas shell. El comando-interprete **shell** es cápaz de realizar múltiples tareas, una de estas, es el administrar el propio sistema.

Cuando se instala un sistema multiusuario que contenga el sistema operativo UNIX/XENIX, existe una consola maestra y esta es controlada por un operador, cuyo objetivo primordial es el de administrar eficientemente los siguientes dispositivos de la computadora :

Terminales
Impresoras
Unidades de cintas y diskettes
Espacio en disco duro
Administar las tareas y distinguir prioridades de los usuarios

.
.
etc

Generalmente la persona asignada a la consola maestra del sistema, no posee los conocimientos adecuados para admistrar eficientemente el CPU, ya que la buena administración, exije diversos conocimientos necesarios, tales como :

El conocimiento del idioma inglés

Una vez que el operador empieza a tener contacto con los manuales referentes al equipo, resulta que la mayoria de estos , estan escritos en ingles.

Esto hace que el aprendizaje sea tedioso, y en ocaciones las ideas de los autores no se aprecian correctamente, como el funcionamiento de los dispositivos del sistema, los errores y soluciones que son emitidos por el CPU y, el estudio de la sintaxis y programacion del shell.

Debido a este problema de conocimiento del idioma ingles, esta tésis muestra de manera fácil y legible los secretos de la programación del **shell**, ofreciendo al operador la gran gama de comandos programados para la buena administración de los dispositivos de la unidad de computo

La comprensión del sistema operativo

El operador de la consola tiene la obligación de conocer el sistema operativo del equipo, ya que dependiendo de la profundidad de estos conocimientos, la administración del equipo será más eficiente.

El sistema operativo UNIX/XENIX, como ya lo mencionamos antes posee una gran variedad de comandos, además de que cada comando tiene aproximadamente desde 0 a 15 opciones, sin mencionar el uso de expresiones regulares y la combinacion de sus opciones en algunos comandos, pero eso no es lo dificil, pues resulta que algunos comandos son lenguajes de programacion de alto nivel y cada comando-lenguaje tiene una sintáxis de programación un poco diferente. Algunos de ellos son :

awK shell bc dc

Y que decir que posee varios editores de texto para diferentes usos y gustos, algunos de ellos son :

vi sed ed

Imaginemos el trabajo del operador al aprenderse por lo menenos 50 comandos y un editor de textos para que mas o menos administre el sistema operativo.

Si suponemos que el sistema cuenta con 30 terminales (una terminal para cada usuario y cada usuario tiene por lo menos 20 archivos), y pedimos al operador tareas como:

a) Que se nos proporcione el espacio en disco de cada usuario, y que nos mencione el tamaño de cada archivo cuando cada usuario tenga más de 1.5 megabytes, así como los archivos que no hayan sido accesados por más de 15 días.

b) El llevar un estricto control de respaldos así como la bitácora de estos, ya sea en cintas ó disKetts.

c) Verificar que nadie borre archivos sin permiso aunque el usuario sea el propietario (evitando el autosabotaje).

d) Asignación de prioridades del uso de la impresora.

e) Cancelación de los reportes no deseados.

f) Apagado del sistema cuando este presente fallas.

Los anteriores incisos son más que suficientes para mantener ocupado al operador en forma constante, pero con la ayuda de la programación del shell, estas tareas se ejecutarán automáticamente mientras que el operador ejecuta otras tareas.

Además de que el comando **shell** es de gran ayuda para los programadores en muchas tareas, algunos de ellos pueden ser:

a) Ayuda en las compilaciones y detección de errores generados por estos, así como de la ejecucion de los programas ya compilados.

b) Pueden simularse pequeñas bases de datos, almacenando en estas informaciones como: agendas telefonicas, diccionarios, glosarios, ayudas en línea.

c) Ayuda en el cambio de archivos entre directorios, así como la transferencia de información de una base de datos a otra, o entre tablas de la propia base de datos (informix-4gl, fox_base etc).

Con el uso de los programas descritos aquí, pueden incluso, construirse muchas aplicaciones ya sea desde procesar grupos de mandatos almacenados en archivos llamados programas Shell o directamente de la línea de comandos.

Es importante señalar que existe poca información acerca de la programación en shell, algunos libros mencionan unos pocos programas en **shell** y generalmente estos ejemplos siempre son los mismos.

Los programas en **shell** aquí presentados, fueron elaborados y enfocados a las necesidades personales de esta empresa y se logró en base a un gran esfuerzo al realizarlos, ya que hasta la fecha no existe información adecuada en los pocos libros que hablan de UNIX.

Se pretende dar conocimientos sólidos de la programación shell, reuniendo la experiencia de varios años en este sistema operativo.

En el capítulo 1 se presenta una breve reseña del sistema operativo UNIX, ya que es importante el conocer sus orígenes y cómo funciona este sistema, así como el conocer a sus diseñadores.

En el capítulo 2 se nos muestran los conceptos básicos que el iniciado debe conocer antes de entrar a la programación del shell, estos conceptos nos enseñan la estructura general de los archivos del sistema operativo y nos involucra indirectamente con algunos comandos.

En el capítulo 3 se presentan las reglas de la programación del **shell**, así como de su sintáxis, es de suma importancia hacer notar que **el shell es estandar para cualquier versión del sistema operativo**, lo único que puede variar es el nombre de algunos comandos o el cuerpo del programa

En el capitulo 4 se proporciona la forma de correcta de utilizar las **expresiones regulares**, que son indispensables en algunos comandos, ya que estos también son indispensables en la programación del shell.

En el capitulo 5 se describe la sintaxis de algunos comandos del sistema operativo, estos son sed, awK, bc, y el du ya que ademas de ser procesadores de textos su uso y programación es complicado

En el capitulo 6 se presentan programas elaborados en el lenguaje **shell**, para que el lector pueda comprender el poderio de este lenguaje, además de que cada programa tiene un uso especifico y se encuentran funcionado en esta empresa a la perfección. Todos los programas aquí presentados estan completos.

En el glosario se definen terminos computacionales para aquellas personas que se inician tanto en programación como en el sistema operativo UNIX/XENIX.

CAPITULO 1

BREVE RESENA DEL SISTEMA OPERATIVO UNIX/XENIX

Antecedentes

UNIX es un sistema operativo para computadoras desarrollado en los Laboratorios Bell, en Nueva Jersey, Estados Unidos. En 1969, un grupo de investigadores se abocó a la tarea de crear un entorno de programación que facilitara sus labores internas de investigación y desarrollo. Con el apoyo de Dennis Ritchie y de otros investigadores, Ken Thompson, creó un sistema operativo de tiempo compartido, pequeño y de propósito general. esta primera versión fue escrita en el lenguaje ensamblador de una minicomputadora PDP-7 que ya no usaban; al año siguiente, Ritchie la instaló en una máquina más moderna una PDP-11, y se dedicó a escribir el compilador para el lenguaje de programación C, que acababan de diseñar. En 1973, Thompson y Ritchie escribierón el núcleo de Unix en lenguje C; con ello terminarón la tradición de escribir sistemas operativos en lenguaje ensamblador y lograrón además, que Unix fuera más portátil y fácil de modificar. Poco después se concedió el permiso para que algunas instituciones no lucrativas tuvieran acceso a Unix en la versión de la PDP-11, que ya era muy popular en universidades e instituciones de investigacion, y comenzó la rápida difusión del sistema en todo el mundo. En la actualidad, el sistema Unix se considera un estándar virtual para computadoras multiusuario, y ha sido adoptado para gran cantidad de máquinas. Existen distintas versiones comerciales del sistema :

- Unix III
- Unix V
- Unix BSD
- Xenix, etcetera

pero todos tiene mucho en común.

El nombre Unix proviene de un juego de palabras acorde con la filosofía de su diseño. En 1965, los Labotatorios Bell y la Compañía General participaron en una proyecto de desarrollo de sistemas operativos, integrado al proyecto MAC del Instituto Tecnológico de Massachusets (MIT), cuyo objetivo era diseñar un gran sistema multiusuario denominado 'Multics' . De esta experiencia se recogierón muchos aspectos que hasta la fecha son importantes en la programación de sistemas, pero el proyecto no culminó. Esto se debió, en parte, a que se trataba de un diseño muy amplio y complejo. Thompson, Ritchie y otros participantes en el proyecto Multics aprendieron la lección y años después bautizaron su nuevo sistema con el nombre de Unix, que tiene una connotación contraria a la de multiplicidad y complejidad Unix y la mayor parte de los sistemas que se ejecutan en él están escritos en lenguaje C, y han servido para demostrar que un sistema operativo interactivo no necesariamente es grande y caro, ya sea en equipo o en la cantidad de código, ya que puede utilizarse en minicomputadoras de costo reducido y el desarrollo inicial de su sistema principal requirió menos de 2 años hombre. En palabras de sus creadores, el objetivo es que el sistema sea simple, elegante y fácil de usar.

Si se recuerda que la función general de un sistema operativo es controlar y dirigir la operación de la computadora, de forma tal que presente una imagén monolítica y virtual ante los usuarios del sistema

de cómputo, estaremos de acuerdo entonces en que dicho sistema es tan importante como las fácilidades físicas y electrónicas de su equipo.

Lo que se espera de un sistema operativo es que sea cápaz de atender la operación concurrente de múltiples pedidos de atención por parte de los procesos que están ejecutándose en la computadora; que pueda mantener toda la operación bajo control sin perder detalle alguno ni permitir que los procesos interfieran entre sí; que logre un aprovechamiento óptimo de los recursos físicos de la máquina (procesador, memoria, periféricos) y, por último, que haga todo esto en forma silenciosa y eficiente. Como es fácil de comprender, son pocos los sistemas operativos que cumplen todos estos requisitos que a veces son incluso contradictorios; no puede esperarse, por ejemplo, que el sistema sea potente, inteligente, eficiente y pequeño al mismo tiempo.

La razón de la creciente popularidad de Unix reside en que logra combinar fácilidad de uso y eficiencia, y en la gran cantidad de ayudas utilerías que tiene para programar. Con Unix es sencillo obtener comunicación y sincronización entre procesos, lo que en otros sistemas operativos requiere de programanción dedicada y exclusiva en los lenguajes de control o incluso, en los más limitados, es virtualmente imposible de lograr.

La filosofía de operación de Unix está basada en el concepto de herramientas de software. Esta visión conceptual pide que las tareas computacionales se construyan paulatinamente (de manera que podría llamarse genética); el sistema aporta un conjunto de operaciones primitivas que el diseñador usa para armar aplicaciones que, una vez hechas, pasan a formar parte del acervo de operaciones básicas. Es decir, con un pequeño número de funciones elementales, pueden configurarse programas y sistemas completos que cumplan funciones específicas.

Entre las caracteristicas del sistema operativo Unix, se encuentran las siguientes:

- Es un sistema operativo multiusuario con capacidad de simular multiprocesamiento y procesamiento no interactivo.

- Esta escrito en un lenguaje de alto nivel : C.

- Dispone de un lenguaje de control programable, llamado shell.

- Ofrece facilidades para la creación de programas y sistemas y el ámbiente adecuado para las tareas de diseño de software.

- Emplea manejo dinámico de memoria (por intercambio o por paginación).

- Tiene capacidad de interconexión de procesos.

- Permite la comunicación entre procesos.

- Emplea un sistema jerárquico de archivos, con facilidades de protección de archivos, cuentas y procesos.

- Usa un manejo consistente de archivos de diversos tipos.

- Tiene fácilidades para redireccionamiento de entradas y salidas.

- Incluye más de un centenar de subsistemas y varios lenguajes de programación.

- Garantiza un alto grado de portabilidad.

- El sistema se basa en un núcleo (conocido como Kernel), que reside permanentemente en la memoria, y que atiende todas las llamadas del sistema, administra el acceso a los archivos y el inicio o la suspención de las tareas de los usuarios.

- Unix permite que los programas sean independientes de los dispositivos periféricos; la salida de cada programa o utilería del sistema pueda dirigirse a archivos de disco, impresoras o terminales, y existe también la posibilidad de comunicacion entre los procesos para crear conjuntos arbitrarios y complejos de procesos concurrentes cooperativos.

La comunicación con el sistema Unix se da mediante un programa especializado de control llamado shell. Este es un lenguaje de control, un intérprete, y un lenguaje de programación, cuyas características lo hacen sumamente flexible para las tareas de un centro de cómputo. Como lenjuaje de programación, abarca estos aspectos:

- Ofrece las estructuras de control básicas :
  - Secuenciación
  - Iteracción condicional
  - Selección y otras

- Paso de parámetros

- Sustitución textual de variables y cadenas

- Comunicación bidireccional entre órdenes del shell

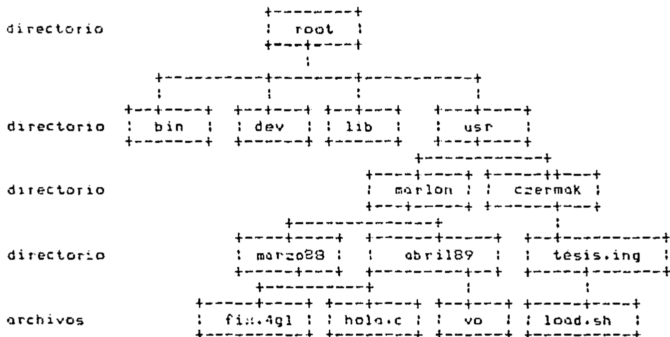
- El shell, permite modificar en forma dinámica las características con que se ejecutan los programas en Unix:

_ Las entradas y las salidas pueden ser redireccionadas o redirigidas hacia archivos, procesos y dispositivos; así mismo, es posible interconectar procesos entre sí, y diferentes usuarios pueden 'ver' versiones distintas del sistema operativo debido a la capacidad del shell, para configurar diversos ambientes de ejecución. Por ejemplo, se puede hacer que un usuario entre directamente a su sesión, ejecute un programa en particular y salga automáticamente del sistema al terminar de usarlo. A veces resulta conveniente programar la primera versión de un sistema de shell, para probarlo en forma interactiva por medio del intérprete. De la misma forma, es sencillo automátizar tareas que suelen hacerse en forma manual tales como agrupamiento de órdenes, ejecución seriada de programas, etc.

Por su parte, el sistema de archivos de Unix; esta basado en un modelo arborescente y recursivo, en el cual los nodos pueden ser tanto archivos como directorios y estos últimos pueden contener a su vez direcctorios o subdirecctorios.

Debido a esta filosofía, se maneja al sistema con muy pocas órdenes que permiten una gran gama de posibilidades, todo el archivo de Unix está controlado por múltiples niveles de protección, que especifican los permisos de acceso al mismo. La diferencia que existe entre un archivo de datos, un programa, un manejador de entrada/salida o una instrucción ejecutable se refleja en estos parámetros, de modo que el sistema operativo adquiere características de coherencia y elegancia que lo distinguen.

La raíz del sistema de archivos (conocido con el nombre de root) se denota con el símbolo /, y de ahí se desprende un conjunto de directorios que contienen todos los archivos del sistema de cómputo. Cada directorio, a su vez, funciona como la subraíz de un nuevo árbol que depende de él y que también puede estar formado por directorios o subdirectorios y archivos. Un archivo siempre ocupará el nivel más bajo dentro del árbol, porque de un archivo no pueden depender otros; si así fuera, sería un directorio. Es decir los archivos son como las hojas de un árbol.



Se define en forma unívoca el nombre de todo archivo (o directorio) mediante lo que se conoce como trayectoria (path name): es decir, el conjunto completo de directorios, a partir de root (/), por los que hay que pasar para poder llegar al directorio o archivo deseado. Cada nombre se separa de los otros por el símbolo /, aunque tan sólo el primero de ellos se refiere a la raíz. Por ejemplo :

/usr/marlon/marzo88/hola.c tiene toda esta trayectoria como nombre absoluto, pero se llama marlon/marzo88/hola.c, sin la diagonal inicial,

si se observa desde el directorio /usr. Para los usuarios que estan normalmente en el directorio /usr/marlon, el archivo se llama **marzo88/hola.c** Asi, también puede existir otro archivo llamado **hola.c**, pero dentro de algún otro directorio y en caso de ser necesario se emplearía el nombre de la trayectoría (completa o en partes, de derecha a izquierda).

Unix ofrece medios muy sencillos para colocarse en diferentes puntos del árbol que forma el sistema de archivos. En esta forma se maneja el sistema completo de archivos y se dispone de un conjunto de órdenes de shell (además de múltiples variantes) para hacer diversas manipulaciones, como crear directorios, moverse dentro del sistema de archivos, copiarlos, etc. Unix incluye, además, múltiples esquemas para crear, editar y procesar documentos. Existen varios tipos de editores, formateadores de textos, macroprocesadores de tablas, procesadores de expresiones matemáticas y un gran número de ayudas y utilerias diversas, que se mencionan más adelante.

A continuacion se describe el modo de funcionamiento de Unix, con base en un modelo estudiado de sistemas operativos que lo dividen en capas jerárquicas para su mejor comprensión.

EL NUCLEO DEL SISTEMA OPERATIVO

El núcleo del sistema operativo Unix (llamdo Kernel) es un programa de aproximadamente 10,000 lineas, escrito casi en su totalidad en lenguaje C, con excepción de una parte del manejo de interrupciones, expresado en el lenguaje ensamblador del procesador en el que opera. Las funciones de núcleo son; permitir la existencia de un ámbiente en el que sea posible atender a varios usuarios y múltiples tareas en forma concurrente, repartiendo al procesador entre todos ellos e intentando mantener en grado óptimo la atención individual.

El Kernel opera como asignador de recursos para cualquier proceso que necesite hacer uso de las fácilidades de cómputo. Es el componente central de Unix y tiene las siguientes funciones :

- Creación de procesos, dar tiempos de atencion y sincronización.

- Asignación de tiempos del microprocesador a los procesos que lo requieren.

- Administracion del espacio del sistema de archivos, que incluye: el acceso, protección y administración de usuarios y entre procesos, y manipulación de E/S y administración de periféricos.

- Supervision de la transmisión de datos entre la memoria principal y los dispositivos periféricos.

El Kernel reside siempre en la memoria central y tiene el control sobre la computadora, por lo que ningún otro proceso puede interrumpirlo sólo pueden llamarlo para que proporcione algún servicio

de los ya mencionados. Un proceso llama al Kernel mediante módulos especiales conocidos como llamadas al sistema. El Kernel consta de 2 partes principales.

a) Sección de control de procesos

Asigna recursos, programas, procesos y atiende sus requerimientos de servicio.

b) Sección de control de dispositivos.

Supervisa la transferencia de datos entre la memoria principal y los dispositivos periféricos.

En terminos generales cada vez que algun usuario oprime una tecla terminal, ya sea para leer o para escribir información del disco magnético, se interrumpe al procesador central y el núcleo se encargará de efectuar la operacion de transferencia. **Cuando se inicia la operación de la computadora**, debe cargarse en la memoria una copia del núcleo, que reside en el disco magnético (operación denominada **bootstrap**). Para ello, se deben inicializar algunas intefaces básicas de hadware; entre ellas, el reloj que proporciona interrupciones periódicas. El Kernel también prepara algunas estructuras de datos que abarcan una sección de almacenamiento temporal para transferencia de información entre terminales y procesos, una sección para almacenamiento de descriptores de archivos y una variable que indica la cantidad de memoria principal. A continuación, el Kernel inicializa un proceso especial, llamado **proceso 0** (cero). En general, los procesos se crean mediante una llamada a una rutina del sistema (**fork**), que funciona por un mecanismo de duplicación de procesos. Sin embargo, esto no es suficiente para crear al primero de ellos, por lo que el Kernel asigna una estructura de datos y establece apuntadores a una sección especial de la memoria, llamada tabla de procesos, que contendrá los descriptores de cada uno de los procesos existentes en el sistema. Después de haber creado el **proceso 0** (cero), se hace una copia del mismo, con lo que se crea el **proceso 1**; éste muy pronto se encargara de dar vida al sistema completo, la activación de otros procesos que también forman parte del núcleo. Es decir, se inicia una cadena de activaciones de procesos, entre los cuales destaca el conocido como despachador (**scheduler**), que es el responsable de decir cuál proceso se ejecutara y cuales van a entrar o salir de la memoria central. A partir de ese momento se conoce el número 1 como proceso de inicialización del sistema, el proceso **init** es el responsable de establecer la escritura de procesos en Unix. Normalmente, es cápaz de crear al menos 2 estructuras distintas de procesos:

1- El modo monousuario
2- El modo multiusuario

Comienza activando el intérprete del lenguaje de control (**shell**) en la terminal principal, o consola, del sistema y proporcionandole privilegios de 'superusuario'. En la modalidad de **un solo usuario** la consola permite iniciar una primera sesión, con privilegios especiales, e impide que las otras líneas de comunicación acepten iniciar sesiones nuevas. Esta modalidad se usa con frecuencia para revisar y reparar sistemas de archivos, realizar pruebas de funciones básicas del sistema y para otras actividades que requieren uso exclusivo de la computadora.

Para el modo multiusuario, se espera pacientemente a que alguien entre en sesión en alguna línea de comunicación. Cuando esto sucede, realiza ajustes en el protocolo de la línea y ejecuta el programa login, que se encarga de atender inicialmente a los nuevos usuarios. Si la clave del usuario y la contraseña proporcionadas son las correctas, entonces entra en operación el programa shell, que en lo sucesivo se encargará de la atención normal del usuario que se dio de alta en esa terminal. A partir de ese momento el responsable de atender al usuario en esa terminal es el intérprete shell, que ofrece todo un amplio conjunto de órdenes y subsistemas para la operación de la computadora, la creación y manipulación de archivos y directorios, la compilación y ejecución de programas y, en general, para mantener la comunicación entre la computadora y los usuarios. Cuando se desee terminar la sesión hay que desconectarse de el shell (y por lo tanto de Unix), mediante una secuencia especial de teclas (usualmente control D) A partir de ese momento la terminal queda disponible para atender a un nuevo usuario.

MANEJO DE MEMORIA

Dependiendo de la computadora en la que se ejecute, Unix utiliza 2 técnicas de manejo de memoria.

a) Swapping o secundaria

b) Memoria virtual

Lo estándar de Unix es un sistema de intercambio de segmentos de un proceso entre memoria principal y memoria secundaria, llamado swapping, lo que significa que se debe mover la imagen de un proceso al disco si éste excede la capacidad de la memoria principal, y copiar el proceso completo a memoria secundaria; es decir, durante su ejecución los procesos son cambiados de y hacia memoria secundaria conforme se requiera. Si un proceso necesita crecer, pide más memoria al sistema operativo y se le da una nueva sección, lo suficientemente grande para acomodarlo. Entonces, se copia el contenido de la sección usada al área nueva, se libera la seccion antigua y se actualizan las tablas de descriptores de procesos. Si no hay suficiente memoria en el momento de la expansión, el proceso se bloquea temporalmente y se asigna espacio en la memoria secundaria. Se copia a disco y posteriormente cuando se tiene el espacio adecuado lo cual sucede normalmente en algunos segundos, se devuelve a memoria principal. Está claro que el proceso que se encarga de los intercambios entre memoria y disco (llamado **swapper**) debe ser especial y jamás podrá perder su posición privilegiada. El kernel se encarga de que nadie intente siquiera interrumpir este proceso, del cual dependen todos los demás. Este es el **proceso** 0 (cero) mencionado antes. Cuando se decide traer a la memoria principal un proceso en estado de 'listo para ejecutar', se le asigna memoria y se copian allí sus segmentos. Entonces, el poceso cargado compite por el procesador con todos los demás procesos cargados Si no hay suficiente memoria, el proceso de intercambio examina la tabla de procesos para determinar cuál puede ser interrumpido y llevado a disco. Una pregunta que surge entonces es : cual de los posibles procesos que están cargados serán desactivados y cambiado a la memoria secundaria ?. Los procesos que se eligen primero son aquellos que están

esperando operaciones lentas (E/S), o que llevan cierto tiempo sin haberse movido al disco. La idea es tratar de repartir en forma equitativa las oportunidades de ejecución entre todos los procesos tomando en cuenta sus historias recientes y sus patrones de ejecución. Otra pregunta es, cúal de todos los procesos que están en el disco será traído a memoria principal ?. La decisión se toma con base en el tiempo de residencia en memoria secundaria. El proceso más antiguo es el que se llama primero, con una pequeña penalización para los grandes. Cuando Unix opera máquinas más grandes, suele disponer de manejo de memoria de paginación por demanda. En algunos sistemas el tamaño de la página en Unix es de 512 bytes; en otros, de 1024. Para reemplazo se usa un algoritmo que mantiene en memoria las páginas empleados más recientemente. Un sistema de paginacion por demanda ofrece muchas ventajas en cuanto a flexibilidad y agilidad en la atención concurrente de múltiples procesos y proporciona, además memoria virtual, es decir, la capacidad e trabajar con procesos mayores que el de la memoria central. Estos esquemas son bastantes complejos y requieren del apoyo de hadware especializado.

MANEJO DEL PROCESADOR

En Unix se ejecutan programas en un medio llamado proceso de usuario. Cuando se requiere una función del Kernel, el proceso del usuario hace una llamada especial al sistema y entonces el control pasa temporalmente al núcleo. Para esto se requiere de un conjunto de elementos de uso interno, que se mencionan a continuación.

- Se conoce como **imagén** a una especie de fotografía del ámbiente de ejecución de un proceso, que incluye una descripción de la memoria, valores de registros generales, status de archivos abiertos, el directorio actual, etc. Una imagén es el estado actual de una computadora virtual dedicada a un proceso en particular.

- **Proceso** se define como la ejecución de una imagen. mientras el procesador ejecuta un proceso, la imagen debe residir en la memoria principal; durante la ejecución de otros procesos permanece en la memoria principal a menos que la aparicion de un proceso activo de mayor prioridad la oblique a ser copiado al disco, como ya se dijo. Un proceso puede encontrarse en uno de dos estados : 1 en **ejecución;** listo para ejecutar, o 2 **en espera.** Cuando se invoca una funcion del sistema, el proceso de usuario llama al Kernel como subrutina. Hoy un cambio de ambientes y, como resultado, se tiene un proceso del sistema En estos 2 procesos son 2 faces del mismo original, que nunca se ejecutan en forma simultánea. Existe una tabla de procesos que contiene una entrada por cada uno de ellos con los datos que requiere el sistema : identificación, direcciones de los segmentos que emplea en la memoria, información que necesita el 'sheduler' y otros. La entrada de la tabla de procesos se asigna cuando se crea el proceso y se libera cuando éste termina . Para crear un proceso se requiere la inicializacion de una entrada en la tabla, así como la creación de segmentos de texto y de datos. Además es necesario modificar la tabla cuando cambia el estado del proceso o cuando recibe un mensaje de otro (para sincronización, por ejemplo). Cuando un proceso termina, su entrada en la tabla se libera y queda disponible para que otro nuevo la utilice. En el sistema operativo Unix los procesos pueden comunicarse internamente entre sí, mediante el envío de mensajes o señales. El

mecánismo conocido como interconexión (pipe) crea un canal entre 2 procesos mediante una llamada a una rutina del Kernel, y se emplea tanto para pasar datos unidireccionalmente entre las imágenes de ambos, como para sincronizarlos, ya que si un proceso intenta escribir en un 'pipe' ocupado, debe esperar a que el receptor lea los datos pendientes. Lo mismo ocurre en el caso de una lectura de datos inexistentes: el proceso que intenta leer debe esperar a que el proceso productor deposite los datos en el canal de intercomunicación.

Entre las diferentes llamadas al sistema para el manejo de procesos que existen en Unix están las siguientes, algunas de las cuales ya han sido mencionadas :

Fork sacar una copia a un proceso

Exec cambiar la identidad de un proceso

Kill enviar una señal a un proceso

signal (especificar la acción por ejecutar cuando se recibe una señal de otro proceso)

exit (terminar un proceso)

Dentro de las tareas del manejo del procesador destaca la asignación dinámica (**sheduling**), que en Unix resuelve el '**sheduler**' mediante un mecanismo de prioridades. Cada proceso tiene asignado una prioridad; las prioridades de los procesos de usuario son menores que la más pequeña de un proceso del sistema. El 'motor' que mantiene en movimiento un esquema de multiprogramación es, por un lado, el conjunto de interrupciones que genera el desempeño de los procesos y, por otro, los constantes recordatorios que hace el reloj del procesador para indicar que se terminó la fracción de tiempo dedicada a cada proceso.

En el sistema Unix, las interrupciones son causadas por lo que se conoce como eventos, entre los cuales se consideran: la ejecución de una tarea de entrada/salida; la terminación de los procesos dependientes de otro; la terminación de la fracción de tiempo asignada a un proceso, y la recepción de la señal desde otro proceso. En un sistema de tiempo compartido se divide el tiempo en un determinado número de intervalos o fracciones y se asigna cada una de ellas a un proceso. Además Unix toma en consideración que hay procesos en espera de una operación de E/S y que ya no pueden aprovechar su fracción. Para asegurar una distribución adecuada del procesador entre los procesos se calculan dinámicamente las prioridades de estos últimos, con el fin de determinar cuál será el proceso que ejecutará cuando se suspenda el proceso activo actual.

MANEJO DE ENTRADAS Y SALIDAS

El sistema de entrada/salida se divide en 2 sistemas complementarios: 1 el estructurado por bloques y 2 el estructurado por espacios. El primero se usa para manejar cintas y discos magnéticos y emplea bloques de tamaño fijo (512 O 1024 bytes) para leer o escribir. El segundo se utiliza para atender a las terminales, lineas de comunicación e impresoras, y funciona byte por byte. En general, el sistema Unix emplea programas especiales (escritos en C) conocidos como manejadores (drivers) para atender a cada familia de dispositivos de E/S. Los procesos se comunican con los dispositivos mediante llamadas a su manejador. Además, desde el punto de vista de los procesos, los manejadores aparecen como si fueran archivos en los que se lee o escribe; con esto se logra homogenidad y elegancia en el diseño.

Cada dispositivo se estructura internamente mediante descriptores llamados número mayor, número menor y clase de bloques o de caracteres. Para cada clase hay un conjunto de entradas, en una tabla, que apunta a los manejadores de los dispositivos. El nomero mayor se usa para signar el manejador correspondiente a una familia de dispositivos; el menor pasa al manejador como un argumento, y este lo emplea para tener acceso a uno de varios dispositivos físicos semejantes. Las rutinas que el sistema emplea para ejecutar operaciones de E/S están diseñadas para eliminar los diferencias entre los dispositivos y los tipos de acceso. No existe distinción entre acceso aleatorio y secuencial, ni hay un tamaño de registro lógico impuesto por el sistema. El tamaño de un archivo ordinario está determinado por el número de bytes escritos en él; no es necesario predeterminar el tamaño de un archivo. El sistema mantiene una lista de áreas de almacenamiento temporal (buffers), asignadas a los dispositivos de bloques. El Kernel usa estos buffers con el objeto de reducir el tráfico de E/S. Cuando un programa solicita una transferencia, se busca primero en los buffers internos para ver si el bloque que se requiere ya se encuentra en la memoria principal, como resultado de una operación de lectura anterior. Si es así, entonces no será necesario realizar la operación física de entrada o salida. Existe todo un mecanismo de manipulación interna de buffers y otro de manejo de listas de bytes, necesario para controlar el flujo de datos entre los dispositivos de bloques y caracteres y los programas que lo requieren. Por último, debido a que los manejadores de los dispositivos son programas escritos en lenguaje C, es relativamente fácil reconfigurar el sistema para ampliar o eliminar dispositivos de E/S en la computadora, así como para incluir tipos nuevos.

MANEJO DE ARCHIVOS Y DE INFORMACION

Como ya se describió, la estructura básica del sistema de archivos es jerárquica, lo que significa que los archivos están almacenados en varios niveles. Se puede tener acceso a cualquier archivo mediante su trayectoria que especifica su posición absoluta en la jerarquía, y los usuarios pueden cambiar su directorio actual a la posición deseada. Existe también un mecánismo de protección para evitar accesos no autorizados. Los directorios contienen información para cada archivo, que consiste en su nombre y un número que el Kernel, utiliza para manejar la estructura interna del sistema de archivos, conocido como nodo-i y hay un para cada archivo, que contiene la información de su dirección en el disco, su longitud, los modos y las fechas de acceso, el autor, etc,. Existe además, una tabla de descriptores de archivos, que es una estructura de datos residente en el disco magnético a la que se tiene acceso mediante el sistema mencionado de E/S por bloques.

El control de espacio libre en disco se mantiene mediante una lista ligada de bloques disponibles. Cada bloque contiene la dirección en disco del siguiente bloque en la cadena. El espacio restante contiene las direcciones de grupos de bloques del disco que se encuentran libres. De esta forma, con una operacion de E/S, el sistema obtiene un conjunto de bloques libres y un apuntador para conseguir más. Las operaciones de entrada y salida en los archivos se llevan acabo con la ayuda de la correspondiente entrada del nodo-i en la tabla de archivos del sistema. El usuario normalmemte desconoce los nodos-i porque las referencias se hacen por el nombre simbólico de la trayectoria. Los procesos emplean internamente funciones primitivas (llamadas al sistema) para tener acceso a los archivos los más comunes son :

open
creat
read
write
seek
close
unlink

Aunque sólo son empleados por los programadores, no por los usuarios finales del sistema. Toda estructura física se maneja 'desde afuera' mediante la filosofía jerárquica de archivos y directorios ya mencionada y en forma totalmente transparente para el usuario. Además, desde el punto de vista del sistema operativo un archivo es muy parecido a un dispositivo.

Las ventajas de tratar a los dispositivos de E/S en forma similar a los archivos normales son múltiples:

un archivo y un dispositivo de E/S se tornan muy parecidos los nombres de los archivos y de los dispositivos tienen la misma sintaxis y significado, así que un programa que espera un nombre de archivo como parámetro puede dársele un nombre de dispositivo, esto logra una interacción rápida y fácil entre procesos de alto nivel.

El sistema Unix ofrece varios niveles de protección para el sistema de archivos, que consisten en asignar a cada archivo el número único de identificación de su dueño, junto con nueve bits de protección, que especifican permisos de lectura, escritura y de ejecución para el propietario, para otros miembros de su grupo (definido por el administrador del sistema) y para el resto de los usuarios. Antes de cualquier acceso se verifica su validez consultando estos bits, que residen en el nodo-i de todo archivo. Además, existen otros 3 bits que se emplean para manejos especiales relacionados con la clave del superusuario.

Otra característica de Unix es que no requiere que el conjunto de sistemas de archivos resida en un mismo dispositivo.

Es posible definir uno o varios sistemas 'desmontables' que residen físicamente en diversas unidades de disco. Existe una orden (mKfs) que permite crear un sistema de archivos adicional, y una llamada al sistema (**mount**) con la que se añade (y otra con la que se desmonta) uno de ellos al sistema de archivos global.

El control de las impresoras de una computadora que funciona con el sistema operativo Unix consiste en un subsistema (**Spool**) que se encarga de coordinar los pedidos de impresión de múltiples usuarios. Existe un proceso de Kernel, que en forma periódica reviza las colas de las impresoras para detectar la existencia de pedidos e iniciar entonces las tareas de impresión. Este tipo de procesos, que son activados en forma periódica por el núcleo del sistema operativo, reciben en Unix el nombre de **daemons** (duendes), tal vez porque se despiertan y aparecen sin previo aviso. Otros se encargan de activar procesos en tiempos previamente determinados por el usuario, o describir periodicamente los contenidos de los buffers de memoria en disco magnético.

LENGUAJE DE CONTROL DEL SISTEMA OPERATIVO

Entre los rasgos definitivos de Unix está el lenguaje de control que emplea, llamado Shell. Es importante analizar 2 funciones más de Shell, llamados redireccionamiento e interconexión.

Asociado con cada proceso hay un conjunto de descriptores de archivos númerados 0, 1 y 2, que se utilizan para todas las transacciones entre los procesos y el sistema operativo. El descriptor de archivo 0 se conoce como la entrada estándar; el descriptor de archivo 1, como la salida estándar, y el descriptor 2, como el error estándar. En general todos están asociados con la terminal de video, pero, debido a que inicialmente son establecidos por Shell, es posible reasignarlos.

Una parte de la orden que comience con el símbolo < se considera como el nombre del archivo que será abierto por Shell y que se asociará con la entrada estándar; en su ausencia, la entrada estándar se asigna a la terminal. En forma similar, un archivo cuyo nombre está precedido por el símbolo > recibe la salida estándar de las operaciones.

Cuando el Shell interpreta la orden **califica < examén > resulta**, llama a ejecución al programa **califica** (que ya debe estar compilado o listo para ejecutar) y detectará la existencia de un archivo que toma el lugar de la entrada estándar (**examen**) y de otro que remplaza a la salida estándar (**resulta**). Entonces pasa como datos de lectura los contenidos del archivo **examen** recién abierto (que debe existir previamente) al programa ejecutable. Conforme el programa produce datos como salida, éstos se guardan en el archivo **resulta** que el shell crea en ese momento.

En la teoría de lenguajes formales desempeñan un importante papel las gramáticas, llamadas de tipo 3 (tambien conocidas como regulares), que tienen múltiples aplicaciones en el manejo de lenguajes. Existen unas construcciones gramáticales conocidas como expresiones regulares, con las que se puede hacer referencia a un conjunto ilimitado de nombres con estructura lexicográfica similar; esto lo aprovecha el shell para dar al usuario fácilidades expresivas adicionales en el manejo de los nombres de los archivos. Así, por ejemplo, el nombre **carta*** se refiere a todos los archivos que comiencen con el prefijo carta y que sean seguidos por cualquier subcadena, incluyendo la cadena vacía; por ello, si se incluye el nombre **carta*** en alguna orden, el shell la aplicará a los archivos **carta, carta1, cartas** etc, y cualquier otro que cumpla con esos especificaciones. En general, en lugares donde se emplea un nombre o una trayectoria, shell permite utilizar una expresión regular que sirve como abreviatura para toda una familia de ellos, y automáticamente repite el pedido de atención para los componentes. Existen además otros carácteres especiales que Shell reconoce y emplea para el manejo de expresiones regulares, lo que proporciona al lenguaje de control de Unix mayor potencia y capacidad expresiva.

En Unix existen también la posibilidad de ejecutar programas sin tener que atenderlos en forma interactiva, sino simulando paralelismo (es decir, atender de manera concurrente varios procesos de un mismo usuario). Esto se logra agregando el símbolo **&** al final de la línea en la que se escribe la orden de ejecución. Como resultado, Shell no

espera que el proceso hijo termine de ejecutar (como lo haría normalmente) sino que regresa a atender al usuario inmediatamente después de haber creado el proceso asincrónico, simulando en esta forma el procesamiento por lotes (batch). Para cada uno de los procesos, Shell proporciona, además, el número de identificación, por lo que si fuera necesario el usuario podría cancelarlo posteriormente, o averiguar el avance de la ejecución.

La comunicación interna entre procesos , el envío de mensajes con los que los diversos procesos se sincronizan y coordinan ocurre mediante el mecánismo de interconexiones (pipes) ya mencionado, que conecta la salida estándar de un programa a la entrada estándar de otro, como si fuera un coducto con 2 extremos, cada uno de los cuales está conectado a su vez a un proceso distinto. Desde Shell puede emplearse este mecánismo con el símbolo ¦ en la línea donde se escribe la orden de ejecución;

(califica < tarea ¦ sorte > lista) &

Se emplean las características de interconexión, redireccionamiento y asincronía de procesos para lograr resultados difíciles de obtener en otros sistemas operativos. Aquí se pide que en forma asincrónica es decir, dejando que la terminal siga disponible para atender otras tareas del mismo usuario, se ejecute el programa **califica** para que lea datos que requiere del archivo **tarea** ; al terminar, se conectará con el proceso **sort** es decir, pasará los resultados intermedios para que continúe el procesamiento y se arreglen los resultados en orden alfabético; al final de todo esto, los resultados quedarán en el archivo **lista**.

Con esta otra orden, por ejemplo

egrep -n 'contrato' ¦ 'empleado' E*

Se busca obtener todos los renglones que contengan las palabras **'contrato'** o **'empleado'** en los archivos de disco cuyos nombres comiencen con la letra 'E' (lo cual se denota mediante una expresión regular). Para lograrlo, se hace uso de una función llamada **egrep**, especial para el manejo de patrones y combinaciones de expresiones regulares dentro de los archivos Los resultados aparecen así :

Emple1:5: en caso de que un empleado decida hacer uso de la fácilidad,

Emple1:7:y el contrato así lo considere:las obligaciones de la

Emple2:9:Cláusula II =;El contrato colectivo de trabajo específico

Emple2:15:Fracción III:El empleado tendrá derecho, de acuerdo con lo.

El tercer renglon, por ejemplo,muestra el noveno renglon del archivo Emple2, que contiene una de las palabras buscadas.

Como Unix fue diseñando para servir de entorno en las labores de diseño y producción de programas, ofrece además de su filosofía un rico conjunto de herramientas para la creación de sistemas complejos, entre las que destaca el subsistema MAKE. Este último ofrece una especie de

lenguaje muy sencillo, con el cual el programador describe las relaciones estructurales entre los modulos que configuran un sistema completo, para que de ahí en adelante make se encargue de matener el sistema siempre al dia. Es decir, si se modifica algún módulo, se reemplaza o se añade otro, las compilaciones individuales, así como las cargas y ligas a que haya lugar, serán realizadas en forma automática por esta herramienta. Con una sola orden, es posible efectuar decenas de compilaciones y ligas predefinidas entre módulos, y asegurarse de que en todo momento se tiene la última versión de un sistema, ya que tambien se lleva cuenta automática de las fechas de creación, modificación y compilación de los diversos modulos; de esta manera, se convierte en una herramienta casi indispensable al desarrollar aplicaciones que requieren decenas de programas que interactúan entre sí o que mantienen relaciones jerárquicas.

Otros comandos interesantes son:

ar. diseñado para crear y mantener bibliotecas de programas que serán luego utilizadas por otros programas para efectuar las funciones ya definidas sin tener que duplicar el código.

awk. Un lenguaje para reconocimiento de patrones y expresiones regulares es decir, generados por una gramática regular o de tipo 3 , útil para extraer información de archivos en forma selectiva.

lex. Un generador de analizador lexicográfico

yacc. Un compilador de compiladores

Estos 2 últimos se emplean como herramientas en la creación de compiladores y procesadores de lenguajes.

La lista completa de funciones, órdenes de subsistemas que forman parte de las utilerías del sistema operativo Unix es realmente grande, e incluye más de un centenar, que se pueden agrupar en los siguientes rubros :

Compilador de compiladores
Ejecución de programas
Fácilidades de comunicaciones
Funciones de control de status
Funciones para control de usuarios
Funciones para impresión
Herramientas de desarrollo de programación
Lenguaje C (funciones y bibliotecas asociadas)
microprocesamiento
Manejo de directorios y archivos
Manejo de gráficos
Manejo de información
Manejo de terminales
Mantenimiento y respaldos
Otros lenguajes algoritmicos integrados
Preparación de documentos

COMENTARIOS FINALES

Un sistema operativo es mucho más que un amplio conjunto de programas; representa, de hecho, la forma que tendrá una computadora ante sus usuarios. Es aquí donde el sistema Unix es especial, pues fue diseñado originalmente con una filosofía muy clara y explícita: servir como marco de referencia para desarrollar software. Esta marca de origén explica, el gran exito de Unix entre la comunidad académica y computacional y su relativamente menor penetración y popularidad en el mercado del procesamiento de datos y la informática comercial. De hecho, el objetivo primario de una computadora y del sistema operativo que la hace ser lo, que es, en el entorno comercial o de producción, es precisamente servir como vehículo para la exploración de sistemas ya creados. Por lo tanto, al usuario de determinado sistema le preocupa más la fácilidad de operación que la posibilidad de crear estructuras computacionales elegantes o complejas.

La corta historia de la computación ha mostrado que en este campo del quehacer humano, como en todos los demas no hay panaceas ni soluciones universales. Aquí han existido también los inevitables intentos por definir la realidad de acuerdo con intereses particulares o de mercado, y al paso de los años los hemos visto fracazar. En el campo de los leguajes de programación, por ejemplo, se ha propuesto que tal o cual lenguaje es el adecuado, Algol y PL/I han sido ejemplos de esta megalomanía, así como mas recientemente lo es Ada, y el futuro próximo depara más revelaciones y sorpresas de este tipo.

La filosofía que subyace al diseño de Unix apunta claramente hacia un campo de aplicacines creativas, pero que requieren conocimiento especializado previo. No es, en efecto, un sistema oscuramente difícil, pero tampoco fue creado pensando en usuarios casuales o poco interesados. Para este tipo de mercado, existen en muchas máquinas de UNIX 'frentes amigables' que guían al usuario mediante menús y pantalla de ayuda, por lo que tampoco quedan excluidas de su espectro de aplicaciones.

Sin embargo, quienes emplean el sistema Unix como herramienta y entorno de creación de programas y sistemas, encuentran en él un campo extremadamente fértil, hasta podríamos decir exuberante, para sus esfuerzos. Esto se debe, como se ha dicho, a la filosofía de herramientas de software con lo cual fue creado. El reconocimiento de la comunidad internacional también llegó ya; la prestigiada Association for Computing Machinery otorgó a Dennis Ritchie y a Ken Thompson el premio 'Alan Turing' de 1983, por su labor en el desarrollo de Unix.

Este premio es considerado el máximo reconocimiento a la calidad académica o profesional en el campo de la computación en el mundo. Los galardonados pronuncian un discurso en la ceremonia de aceptación y una versión adaptada se pública después en la revista oficial de la asociación Comunications of the ACM. Los artículos de Ritchie y Thompson aparecieron en el número de agosto de 1984. La editorial Adisson-Wesley publicó en 1987, el libro ACM Turning Award Lectures: The first Twenty Years, que contiene los artículos de los premiados

entre 1966 y 1985. Y se trata prácticamente de un directorio de los principales investigadores en computación, ya que aparecen entre otros, los nombres de Knut, Dijkstra, Bakus, Hoare, Wirth, Wilkes, Mc Carthy, Simons, Rabin, Iverson y Codd.

De hecho Unix no es un sistema operativo monolítico, como casi todos los demás, sino que está compuesto de un pequeño nucleo y de un conjunto (que a veces parece casi ilimitado) de rutinas y operadores, que literalmente crean atmósfera que envuelve a la programación y al diseño de sistemas. Es más, el campo conocido como ingeniería del software esto es, la creación de programas y sistemas con un método científico y no basado en el método de ensayo y error se ve en Unix, casi la culminación de sus expectativas, puesto que el diseñador de sistemas se rodea de herramientas de todo tipo, que van desde comparadores de archivos y contadores de palabras hasta subsistemas para la generación de reconocedores de lenguajes.

Todo esto, además, está ligado al hecho de que Unix es un sistema operativo relativamente caro en recursos: requiere de un sistema de disco rígido, rápido y eficaz; de velocidad del procesador central, y de manejadores eficaces de entrada y salida. No todo esto es accesible en las computadoras personales, ni estas fueron diseñadas para ello. No es de extrañar pues que Unix sea menos popular que el sistema estándar en computadoras personales (MS-DOS ahora OS/2 en el futuro), o que su filosofía de uso sea otra.

Nada de lo anterior, por supuesto, impide que Unix sea un vehículo óptimo para la productividad, tanto operativa como de diseño, de hecho, cuando uno ha trabajado con Unix, ya no desea sentirse desprotegido.

* **Unix** es una marca registrada de los laboratorios **AT&T Bell**
* **Xenix** es una marca registrada por **Microsoft**

CAPITULO 2

CONCEPTOS BASICOS

CONCEPTOS BASICOS

Antecedentes

En esta parte se introducen los conceptos básicos que se necesitan para saber usar el sistema operativo UNIX/XENIX. Después de leer este cápitulo se podran entender como los archivos, directorios y dispositivos del sistema son organizados y nombrados, como los comandos son introducidos, y cómo la entrada y la salida pueden ser manipuladas.

ARCHIVOS

Un archivo es una coleccion de bytes guardados electronicamente en una unidad de disco o cinta. Los archivos son de 3 clases diferentes :

ARCHIVOS ORDINARIOS
DIRECTORIOS DE ARCHIVOS
ARCHIVOS ESPECIALES DE DISPOSITIVOS

ARCHIVOS ORDINARIOS

Contiene caracteres organizados conjuntamente en una forma escogida por el usuario o por una oplicación particular de un programa. Cada archivo ordinario tiene los siguientes atributos :

- nombre del archivo (no necesariamente es unico)
- nodo-i numero unico dado por el sistema
- tamaño en bytes
- la fecha de creación
- la fecha y hora de la última modificacion
- la fecha y hora del último acceso
- un determinado permisos de acceso

Los permisos de acceso aseguran la privacidad y la seguridad a los archivos. El dueño provee los permisos de lectura, de escritura y ejecucion, así como el control de acceso. el cual lo puede llevar acabo : el propietario, un grupo de usuarios y cualquier otro usuario. Por default, el propietario de un archivo es su creador y como tal el tiene todos los permisos, otros usuarios pueden leer los archivos de otro propietario pero no pueden escribir sobre él.

DIRECTORIOS DE ARCHIVOS

Contienen el nombre y el nodo-i de cada archivo o directorio residente dentro del directorio dado, un nodo-i es un número único asociado con el archivo dado. Todos los archivos que estan en el sistema tienen un nodo-i. Como en los archivos ordinarios, a los directorios se les puede asignar los permisos apropiados de acceso para asegurar la privacidad y la seguridad. Por ende el propietario de un directorio puede leer, crear o remover archivos dentro de su directorio. Similarmente un usuario puede leer archivos dentro de el directorio de otro pero no puede añadir o remover ningun archivo.

ARCHIVOS ESPECIALES DE DISPOSITIVOS

Corresponden a los dispositivos físicos tales como : el disco duro, las unidades de diskettes y cintas, las impresoras, las terminales y sistemas de memoria.

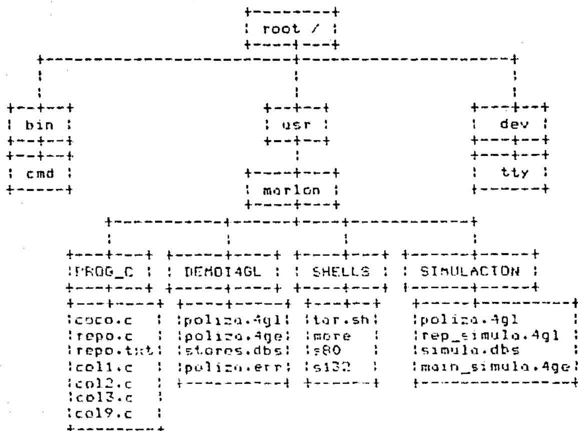
ESTRUCTURAS DE LOS DIRECTORIOS

El sistema operativo organiza todos los archivos dentro de un directorio estructurado interactivamente, el usuario de cada sistema tiene un directorio personal usualmente conocido como : directorio de casa, que puede contener subdirectorios a través del cual el usuario continua con sus derechos de propietario. Un diágrama de un directorio típico de un usuario es el siguiente :

```
                              +-----+
                              : usr :
                              +--+--+
                                 :
                            +----+---+
                            : marlon :
                            +----+---+
        +-----------+-------+----+------------+
        :           :            :            :
  +---+---+  +-----+-----+  +----+---+  +------+-----+
  :PROG_C :  : DEMO14GL  :  : SHELLS :  : SIMULACION :
  +---+---+  +-----+-----+  +----+---+  +------+-----+
  +---+----+  +----+-----+  +---+--+    +-----+---------+
  :coco.c  :  :poliza.4gl:  :tar.sh:    :poliza.4gl     :
  :repo.c  :  :poliza.4ge:  :more  :    :rep_simula.4gl :
  :repo.txt:  :stores.dbs:  :s80   :    :simula.dbs     :
  :col1.c  :  :poliza.err:  :s132  :    :main_simula.4ge:
  :col2.c  :  +----------+  +------+    +---------------+
  :col3.c  :
  :col9.c  :
  +--------+
```

ORGANIZACION DE LOS ARCHIVOS DEL SISTEMA

Es una organización determinada de archivos. En el sistema operativo los archivos del sistema son un camino para accesar todos los recursos de la máquina y estan organizados interactivamente. Un ejemplo de esto es:



CONVENCIONES

Cada archivo, directorio o dispositivo en el sistema operativo tienen un nombre y una ruta especifica, la ruta es un mapa de la localización de los archivos o directorios en el sistema. La ruta completa es única dentro del sistema entero, los nombres de los archivos son únicos solamente dentro de su directorio y pueden no ser únicos en los subdirectorios ejemplo :

archivo	ruta
poliza.4gl	/usr/marlon/SIMULACION/poliza.4gl
poliza.4gl	/usr/marlon/DEMOI4GL/poliza.4gl

NOMBRES DE ARCHIVOS

Es una secuencia de 1 a 14 carácteres; cada archivo, directorio y dispositivo en el sistema tiene un nombre. El nombre del archivo es el único identificador del contenido del directorio. Mientras 2 archivos esten en el mismo directorio estos no pueden tener nombres iguales, pero los nombres en diferentes directorios o subdirectorios si pueden ser identicos.

Así mismo, se puede usar casi cualquier carácter en un nombre de archivo, excepto los carácteres de control que tienen funciones especiales y no pueden ser usados en los nombres de los archivos, estos son:

(-)
(?)
(*)
([y])
(",' ,`)
(/ , \ )

Y los carácteres de control

NOMBRES DE RUTA

Es una secuencia de nombres de directorios seguidos por un simple nombre de un archivo, donde cada nombre de archivo o directorio es separado por una diagonal (/); i.e:

/bin/cmd

Un nombre de ruta que empieza con / , es llamado nombre de ruta lleno, porque éste especifica que un archivo debe ser buscado desde el directorio root.

MUESTRAS DE NOMBRES DE RUTAS

/	Este es el nombre absoluto de la ruta de el directorio de root del sistema de archivos entero.
/bin	Este directorio contiene comandos usados frecuentemente.
/usr	Este directorio contiene los directorios personales de cada usuario del sistema.
/usr/bin	Este directorio contiene los comandos usados frecuentemente que no estan en /bin.
/dev	Directorio que contiene archivos correspondientes a los dispositivos fisicos del sistema.
/dev/console	Nombre de la terminal maestra del sistema.
/dev/tty	Nombre de las terminales de los usuarios.

/lib — Directorio que contiene archivos usados por algún comando estandar.

/tmp — Directorio que contine los archivos temporales dañados.

Cada usuario reside en un directorio llamado directorio actual y todos los archivos y directorios tienen un directorio origén (excepto root). El directorio de origen es uno inmediatamente arriba del directorio actual. El sistema de archivos provee notas taquigráficas especiales para el directorio origén y el directorio actual.

. (punto) Este nombre taquigráfico es del directorio actual, es decir si estoy en el directorio :

.. Nombre taquigráfico del directorio origen.

../.. Nombre taquigráfico del directorio que esta 2 niveles arriba del directorio actual, en resumen:

ruta	tipo de directorio	nombre taquigráfico
/usr/marlon	actual	. (punto)
/usr	origén	..
/	origen de /usr	../..

CARACTERES ESPECIALES

El sistema operativo permite usar caracteres especiales que especifiquen una secuencia de nombres que contengan patrones comunes.

Caracteres	tipo de busqueda
*	cero o mas caracteres
[ ]	cualquier caracter que este dentro
?	cualquier simple caracter

El asterisco (*) puede ser usado en cualquier parte del nombre de un archivo y, puede ocurrir en varios tiempos. Un asterisco busca un juego de carácteres en todos los archivos que no contengan / o que empiecen con períodos.

Si estoy en el directorio /usr/marlon/PROG_C

y listo lo siguiente :

lf *o.c	lf r*	lf *co*	lf *txt
coco.c	repo.c	coco.c	repo.txt
repo.c	repo.txt	col1.c	
		col2.c	
		col3.c	
		col9.c	

juego con lo que este en el interior de los mismos. Además puede efectuar busquedas por rangos consecutivos de números y letras.

si estoy en el directorio /usr/marlon/PROG_C

y listo lo siguiente :

lf col[129]*	lf col[1-9]*	lf *co[a-z]*
col1.c	col1.c	coco.c
col2.c	col2.c	col1.c
col9.c	col3.c	col2.c
	col9.c	col3.c
		col9.c

El signo de interrogacion (?) busca un solo patrón o carácter y su funcionamiento es identico al del asterisco.

Cuando usamos caracter especial (*, [ ], ?) literalmente encerramos el argumento entero en unas comillas cerrados.

COMANDOS

Esta seccion describe la linea de comandos y su sintaxis general, cuando se introducen comandos

LINEA DE COMANDOS

Una linea de comandos es una serie de carácteres que se envian a un shell cuando se escribe un comando y presiona **Retn**, el shell lee la linea de comando y ejecuta los comandos apropiados, pero si se presiona la tecla de interrupcion **BreacK/Del** antes de presionar enter la linea de comando es abortada.

Se pueden útilizar multiples comandos en una linea de comando siempre y cuando estos sean separados por punto y coma (;). Para ejecutar un proceso en background se añade un ampersad (&) al final de

la linea de comandos, de este modo la ejecución es similar a los procesos en lotes de otros sistemas. La principal ventaja de ejecutar los comandos en background es que usted puede ejecutar otras tareas desde su terminal en primer termino mientras los comandos en background se ejecutan.

SINTAXIS

La sintaxis general de los comandos es la siguiente :

comando [opciones o switches] [argumentos] [archivo(s)]

Por convención, los nombres de los comandos son en **minúsculas.**

Opciones - van precedidas de un guion (-) al inicio de las letras que representan las opciones

Argumentos - proveen información adicional de ayuda de ese comando cuando esta ejecutando su tarea.

Archivo(s) - son los que especifican los nombres de archivo(s) requeridos por el comando.

ENTRADAS Y SALIDAS

El sistema operativo asume que la terminal lee las entradas desde el teclado y que las salidas son por video. Pero se puede redireccionar la entrada/salida de un comando, la entrada puede ser desde un archivo, un comando (en lugar del teclado) y la salida puede ser direccionada a un archivo, impresora, otros comandos (en lugar del video). Además tambien se pueden crear pipes (conductos), los cuales permiten que la salida de un comando sea la lectura de otro comando.

REDIRECCIONAMIENTO

En el sistema operativo, un archivo puede sustituir a la terminal ya sea para la entrada y/o salida, los símbolos de los redireccionamientos son :

para la salida	>
para la salida y añadir al final de un archivo	>>
para la entrada	<

El símbolo < significa que toma la entrada para un programa desde el siguiente archivo, en vez de la terminal, así usted puede hacer un archivo de comentarios llamado **carta.txt**, entonces se puede enviar dicho archivo como entrada de un comando que cuente el número de líneas de el archivo **carta.txt**

```
wc -l < carta.txt
```

PIPES (CONDUCTOS)

Una de las mejores innovaciones del sistema Unix/Xenix es el pipe. Este conduto tolera que se conecte la salida de un comando a la entrada de otro, de tal manera que los 2 corren en una secuencia llamada linea de conductos y su símbolo es (¦)

CAPITULO 3

EL SHELL

EL SHELL

Antecedentes

Cuando un usuario esta en el sistema operativo, éste se comunica con el comando interprete **shell** , sh, que es un verdadero y poderoso comando-lenguaje, cada vez que es llamado un shell, este tiene una funcion : leer y ejecutar comandos desde la entrada estandar. El shell proporciona al usuario un lenguaje de alto nivel con el que se puede comunicar con el sistema operativo para ejecutar tareas especificas. Los comandos que normalmente tiene el sistema operativo estan escritos en el lenguaje **C**, pero con el lenguaje de programacion shell estos comandos pueden ser escritos con unas cuantas lineas.

Con el sistema operativo Unix/Xenix y el shell los comandos puede ser :

- cambiar y formar nuevos comandos
- pasados como parámetros posicionales
- añadidos o renombrados por el usuario
- ejecutados dentro de ciclos condicionales
- ejecutados en forma local sin afectar los comandos de otros usuarios que tengan el mismo nombre de comandos
- ejecutados en background
- redireccionados

Conceptos básicos

El shell es un programa que busca ser ejecutado al estar dentro del sistema, además el shell interactua con el usuario (a través del prompt), interpreta y ejecuta comandos introducidos por el teclado. Cuando se esta entrando al sistema se es asignado un shell desde el cual se pueden ejecutar los comandos, este shell es una copia del comando interprete shell del sistema.

COMANDOS

Un camino muy comun que usa el shell es escribiendo comandos simples desde el teclado, cuando es introducido uno de ellos, el comando es enviado al shell, el cual busca en los directorios en los que pueda encontrarse dicho comando, cuando lo encuentra, se copia el archivo (comando) y se envia de regreso al lugar donde fue solicitado y se ejecuta.

COMO BUSCA LOS COMANDOS EL SHELL

El shell normalmente busca los comandos en 3 directorios del sistema :

1- en el directorio actual
2- en el directorio /bin
3- en el directorio /usr/bin

Por ejemplo los comandos **ps** y **ua** se encuentran en los archivos **/usr/bin/ua** y **/bin/ps** respectivamente, un camino más complejo puede ser proporcionado para su localizacion, ya sea un comando relativo en el directorio actual del usuario o un comando con un pathname absoluto. Si el nombre del archivo empieza con / (como /bin/ps) el comando es ejecutado como es nombrado.

Este mecanismo proporciona al usuario un camino conveniente para ejecutar comandos públicos y comandos en, o cerca del directorio actual también que, la capacidad para ejecutar cualquier comando accesible, a pesar de esta localización en la estructura de archivos del sistema. Porque el directorio actual es usualmente donde se inicia la primera busqueda, alguien puede poseer una versión privada de un comando público , puede afectar a otros usuarios. Similarmente la creacion de un nuevo comando público no afecta a usuarios que esten alrededor y tienen comandos privados con el mismo nombre, para cambiar la secuencia de la busqueda de directorios. se puede cambiar la varable del shell : **PATH.**

USANDO METACARACTERES EN LA LINEA DE COMANDOS

Los argumentos de los comandos son muchas veces nombres de archivos algunas veces un grupo de archivos relacionados tienen similares, pero no identicos nombres de archivos, el shell proporciona un determinado conjunto de carácteres especiales llamandos metacarácteres, estos fácilmente especifican un grupo de nombres similares de unos archivos que un argumento de comando. Se puede sustituir un metocaracter por una porción de una linea de comando y el sistema hallará todos los archivos que hagan juego con la busqueda especificada.

El sistema usa los siguientes metacarácteres

***** cualquier cadena (string) (incluyendo una cadena nula, excepto los strings que comiencen con un periodo (.)).

? un solo carácter

[...] cualquier carácter encerrado en los corchetes

[x-y] o [2-29] cualquier carácter que este dentro del rango especificado dentro de los corchetes

ejemplos

*	todos los nombres del directorio actual
temp	todos los nombres que contengan la subcadena temp
[a-z]*	todos los nombres que empiecen desde la a hasta z
*.c	todos los nombres que terminen con .c
/usr/bin/?	todos los nombres de un solo carácter

El juego de patrones tiene algunas restricciones. Si el primer caracter de un nombre de archivo es un punto (.) este puede hacer juego solamente por un argumento que literalmente empiece con un punto (.) , si el patrón no hace juego con cualquier nombre de archivo, entonces el patrón mismo es impreso a la salida como el resultado del juego.

Notese que los nombres de los directorios no pueden ser contenidos en los siguientes carácteres.

* ? [ ]

Si estos carácteres son usados, entonces una infinita recursión puede ocurrir durante la busqueda del patrón.

MECANISMOS DE LAS COMILLAS

Los carácteres <, >, *, ?, [y] tienen significados especiales para el shell, quitar el significado especial de estos caracteres se requieren alguna forma de comillas. estas son las comillas cerradas (' o las dobles comillas (") . Las comillas abiertas (`) son usadas solamente para sustitucion de comandos en el shell y no quitan el significado especial de cualquier carácter.

Todos los carácteres dentro de las comillas son interpretados literalmente así.

QUIENES='who|wc -l'

Se asigna la cadena who|wc -l a la variable QUIENES y no el resultado de su ejecución.

Dentro de las dobles comillas, ciertos carácteres retienen su significado especial, mientras los otros carácteres son interpretados literalmente.

Los carácteres que retienen su especial significado son :

$ \ ` " Así dentro de las dobles comillas las variables son desplegadas y los comandos sustituidos toman lugar (ambos topicos son discutidos más adelante). Sin embargo cualquiera de los comandos dentro de un comando sustituido son inalterados por las dobles comillas, estos carácteres son semejantes como: * ; retienen su significado especial.

Para quitar el significado especial de los signos $ ` " dentro de las dobles comillas precedance estos carácteres con un \.

Fuera de las dobles comillas si precedemos un caracter con \ es equivalente a poner una comilla cerrada al rededor del caracter.

Una \ seguida por otraa causa que la nueva linea sea ignorada y es equivalente a un espacio. La nueva linea precedida por \ es por lo tanto util para dejar espacio y continua una larga linea de comando.

La suguiente lista muestra como el shell hace uso de los comillas

Entrada	El shell lo interpreta como
'`'	` una comilla abierta
\`	` una comilla abierta
\sam	
"\'"	' una comilla
"\$"	$
"\""	" una doble comilla
'"'	" una doble comilla
'`sam`'	la palabra `sam`
"`sam`"	la palabra `sam`
sam reed	2 palabras sam reed
"sam reed"	1 palabra sam reed
'sam reed'	1 palabra sam reed
'sam * reed'	1 palabra sam * reed
`echo sam`	1 palabra sam (ya que echo es como un print y las comillas simples abiertas ejecutan comandos)
`date`	la fecha actual del sistema

REDIRECCIONANDO LA ENTRADA Y LA SALIDA

En general, la mayor parte de los comandos no saben si la entrada o salida, va o viene a la terminal o a un archivo. Unos pocos comandos varian estas acciones, dependiendo de la naturaleza de la entrada o salida, algunos por eficiencia y otros para evitar acciones inservibles.

Cuando un comando empieza su ejecucion, este asume que la entrada, la salida y el error estandar estan abiertas y disponibles. Asociados con cada uno de estos, existe un numero descriptor de archivo :

0	entrada estandar
1	salida estandar
2	Diagnostico de la salida o error estandar

Un proceso hijo normalmente hereda estos archivos desde su proceso padre. Estos 3 archivos son inicialmente conectados a la terminal (0 para el teclado, 1 y 2 para la video), el shell permite que los archivos puedan ser redireccionados a otra parte antes de que el control sea pasado a un comando invocado.

ENTRADA Y SALIDA ESTANDAR

Un argumento para el shell de la forma <file abre el archivo file como entrada , la forma >file abre el archivo file como salida (en caso de salida destruye previamente el contenido del archivo file), la forma >>file direcciona la salida estandar al final del archivo file proveniendo un camino para añadir al final del archivo sin destruir el contenido existente de éste . En cualquiera de los 2 casos de salida, el shell crea el archivo si este no existiese.

$ >file

Crea el archivo vacio file

$ date >> file

Añade la fecha del sistema al archivo file

Tal redirección de los argumentos son solamente sometidos a una variable y un comando de sustitución, ni la interpretación de un blanco ni algun juego con el patron de busqueda de los archivos ocurren después de estas sustituciones, este medio es.

$ echo 'esta es una pregunta'> *.gol

Esto produce un archivo de una linea llamado *.gol y no inserta la cadena 'esta es una pregunta' a todos los archivos que teminen en *.gol

Recuerde que los caracteres especiales no son expandidos en argumentos de redireccionamiento, esto es, la redirección de argumentos son buscados por el shell antes de que el patrón de busqueda sea reconocido y la expansión tome su lugar.

Similarmente un mensaje de error es producido por el siguiente comando :

$ cat < ?

El error es debido a que un archivo no debe llevar el nombre ? ya que es un carácter especial.

DIAGNOSTICO DE SALIDA (ERROR ESTANDAR)

El diagnostico de la salida desde el sistema de comandos es normalmente dirigido al archivo asociado con el archivo descriptor 2 (este es muchas veces necesario para una salida de archivo de error que es diferente desde la salida estandar, así estos mensajes de errores obtenidos no son perdidos), se pueden redireccionar estas salidas de errores a un archivo cercano inmediatamente preparando el número del archivo descriptor (2 en este caso) a cualquier símbolo de redireccionamiento de salida (> o >>). La siguiente línea añade los mensajes de error desde el comando cc (para compilar en C) al archivo llamado errores.

$ cc tablas.c 2>>errores

No debe existir espacio entre el número de archivo descriptor y el símbolo de redireccionamiento o el número que puede ser pasado como un argumento del comando.

Se puede redireccionar la salida asociada con cualquiera de los primeros 10 archivos descriptores (númerados del 0 al 9). Por ejemplo si el comando cmd es puesto a la salida sobre el archivo descriptor 9, entonces la siguiente línea se dirige a la salida del archivo file.

$ cmd 9>file

Un comando muchas veces genera salida estandar y errores de salida y pueden igualmente tener algunas otras salidas, quiza un archivo de datos en este caso, uno puede redireccionar independientemente todas las diferentes salidas, suponga por ejemplo : El comando cmd dirige esta salida con el archivo descriptor 1 al archivo file , el error de salida con el archivo descriptor 2 al archivo errores y establecer un archivo bitácora con el archivo descriptor 9.

$ cmd >file 2>errores 9>bitácora

LINEAS DE COMANDOS Y PIPES (CONDUCTOS)

Una sentencia de comandos separados por la barra vertical (|) constituye un pipe (conducto). Cada comando en un pipe es corrido como un proceso separado conectado a los comandos vecinos por un pipe, esto es, la salida de cada comando (excepto el último) llega a ser la entrada del siguiente comando en la línea. Un filtro es un comando que lee la entrada estandar y transforma a esta en algun camino, entonces escribe este como la salida estandar. Un pipe normalmente consiste en una serie de filtros, aunque los procesos en pipe son permitidos su ejecución en parálelo, cada programa necesita leer la salida de este predecesor. Muchos comandos operan sobre líneas individuales de texto, leyendo una línea, procesando estas o en ciclos que regresan para más lecturas. Algunos deben leer largas cantidades de datos antes de producir salida; el comando sort es un ejemplo de un caso extremo que requiere leer todas las entradas que deben ser leidos antes de que cualquier salida sea producida. La línea siguiente es un ejemplo típico de un pipe.

$ nroff -mm carta | lpr

En el ejemplo anterior, nroff es un formateador de textos del archivo carta y lpr es la impresora actual, la bandera -mm indica una de las comúnmente opciones de formateo usadas y carta, es el nombre del archivo a ser formateado.

Los siguientes ejemplos ilustran la variedad de efectos que pueden ser obtenidos por combinación de unos pocos comandos.

who
nos da la lista de usuarios que estan trabajando en el sistema

who >> quienes
añade la lista de los usuarios al final del archivo quienes

who | wc -l
número de usuarios que esta trabajando en el sistema

who | pr
lista paginado de los usuarios

who | sort
lista alfabetizada de los usuarios que estan trabajando en el sistema

who | grep ro
lista de los usuarios que contengan la subcadena ro

who | grep ro | sort | pr
lista paginada y alfabetizada de los usuarios que contengan la subcadena ro

{ date;who | wc -l;} >>rool
la fecha actual, seguida por el número de usuarios que estan trabajando en el sistema.

este seguro de poner un espacio despues de la llave { y un punto y coma antes de la llave }

who | sed -e 's/ .*//' | sort | unique -d
imprime los nombres de los usuarios que estan en el sistema más de una vez, notese que el uso del comando sed como un filtro para remover los carácteres sobrantes de cada línea (el .* en el comando sed es precedido por un espacio).

El comando who por si mismo no produce todos estos resultados con sus opciones, estos son obtenidos combinando el comando who con otros comandos.

Como un ejercicio remplaze who | con el archivo </etc/passwd en los ejemplos para ver como un archivo puede ser usado como un dato fuente en algun camino.

Note que la redirección de argumentos puede aparecer en cualquier parte sobre la línea de comando. Un ejemplo sería :

$ **<cartas >buzon sort | pr** es lo mismo que :

$ **sort | pr <cartas >buzón**

La primera sentencia dice :

La salida de el archivo cartas es la entrada de el archivo buzon, que se alfabetiza con el comando sort y se página con el comando pr.

la segunda sentencia dice :

alfabetiza y página el contenido del archivo cartas e insertaló en el archivo buzon.

COMANDOS DE SUSTITUCION

Cualquier linea de comando puede ser puesta dentro de comillas abiertas (`...`). La salida del comando se remplaza en la linea de comando entrecomillado. Este concepto es conocido como comando de sustitucion, el comando o comandos encerrados entre comillas abiertas son primero ejecutados por el shell y entonces esa salida remplaza a la expresión entera que esta dentro de las comillas abiertas. Esta caracteristica es obtenida usando la asignación de valores asignados a las variables del shell (estas serán vistas más adelante).

El siguiente comando asigna la cadena representativa de la fecha actual a la variable fecha.

$ fecha=`date`

El siguiente comando conserva el numero de usuarios que estan trabajando en el sistema :

$ usuarios=`who : wc -l`

Cualquier comando escribe su salida a la salida estandar si no es especificado otra.

Dentro de las comillas simples pueden ser puestos otros juegos de comillas abiertas antecediendo \.

si el directorio actual es **/usr/marlon**

$ mensaje=`echo el directorio actual de trabajo es \`pwd\``

y cuando se despliege la variable mensaje su salida será :

el directorio actual de trabajo es **/usr/marlon**

VARIABLES DEL SHELL

El shell tiene diversos mecánismos para crear variables, una variable es un nombre que representa el valor de una cadena. Ciertas variables son referidas como parámetros posicionales, estas son las variables que normalmente son determinadas sobre la línea de comandos. Otras variables del shell son simples nombres que el usuario o el shell mismo puede asignar valores de cadena.

PARAMETROS POSICIONALES

Cuando un procedimiento del shell es invocado, el shell implicitamente crea parámetros posicionales, el nombre del procedimiento del shell mismo es la posicion cero de la linea de comando y es asignado al parámetro posicional **$0**, el primer argumento de la linea de comando es **$1**, el segundo **$2** etc.

Por ejemplo el siguiente scrip en shell, usa un ciclo directo, donde la linea de comando es cambiada.

```
while test '$1'
do
  case $1 in
    -a) A=aopción
        shift
        ;;
    -b) B=bopcion
        shift
        ;;
    -c) C=copcion
        shift
        ;;
    -*) echo 'Mala opcion'
        exit 1
        ;;
     *) procesando el resto de archivos
  esac
done
```

Uno puede explicitamente forzar los valores dentro de estos parámetros posicionales. Usando el comando set, por ejemplo.

$ set abc def hij

Asigna la cadena **abc** al parámetro posicional **$1**, **def** al parámetro posicional **$2**, **hij** al parámetro posicional **$3** y **$0** es el nombre del comando set .

VARIABLES DEFINIDAS POR EL USUARIO

El shell tambien reconoce variables alfanumericas el cual un valor de cadena puede ser asignado. Un ejemplo de asignación tiene la siguiente sintáxis :

letras=cadena

Después de esto, **$letras** reproduce el valor que se le asigna a la variable **letras**, un nombre es una secuencia de letras, digitos y (_), estos nombres empiezan con una letra o un (_) y no debe existir espacios alrededor del signo =, en una sentencia de asignacion notese que los parámetros posicionales pueden no aparecer en el lado izquierdo de la sentencia de asignación, ellos pueden ser determinados solamente como se describe en la sección previa.

Más de una asignación puede aparecer en una sentencia pero, cuidado

El shell ejecuta las asignaciones de derecha a izquierda. Así la siguiente línea de comando resulta en la variable cz el valor abc.

El siguiete ejemplo de asignacion es:

```
$  cz=$w w=abc
```

Dobles comillas alrededor del lado derecho, asignaciones de espacios, tabuladores, comas y lineas nuevas en una cadena, asi como variables de sustitucion y parámetros posicionales son permitidos ocurren.

```
MAIL=/usr/mail/gas
echo_var="echo $1 $2 $3"
estrellas=***
astericos='$estrellas'
```

En el ejemplo anterior la variable echo_var tiene el valor de la cadena que consiste de los valores de los parámetros posicionanes $1, $2, $3 separados por espacios, más la cadena echo. En la variable estrella las dobles comillas no son necesarias alrededor de los *, porque el patron de juego (expansion de los asteriscos, las comillas y los corchetes) no son aplicados en este caso.

Notese que el valor de $asteriscos no es la cadena *** porque los comillas cerrados inhiben esta sustitucion, es decir el valor de $asteriscos es la cadena $estrellas.

En asignaciones los espacios no son reinterpretados despues de la variable de sustitución. En el ejemplo posterior $primero y $segundo toman el mismo valor.

```
$  primero='una cadena con espacios'
$  segundo=$primero
```

Para concatenar el valor de una variable se puede encerrar el nombre de la variable entre llaves { y }, en particular si el carácter inmediatamente seguido por el nombre de la variable es una letra o un digito o un (_) entonces las llaves son requeridas. Por ejemplo :

```
$  agp='esto es un encadena'
$  echo "${agp}miento de cadenas"
```

Cuando esto es ejecutado el comando echo despegara :

esto es un encadenamiento de cadenas

Si las llaves { } no son usadas el shell substituye un valor nulo

```
$  echo "$agpmiento de cadenas"
```

La salida será : **de cadenas**

Las siguientes variables son mantenidas por el shell, algunas de

ellos son determinadas por el shell y todos pueden ser cambiadas por el usuario.

HOME Especifica el nombre del directorio login del usuario, esto es, el directorio de casa . Esta variable es inicializada por el programa login, el comando **cd** sin argumentos se mueve al directorio contenido en **$HOME** Usando esta variable, ayuda a guardar nombres de rutas (pathname) llenos fuera de procedimientos del shell, esto es de gran beneficio cuando los nombres de ruta son cambiados.

IFS Especifica cuales carácteres son campos separadores internos, estos son los que el shell usa durante la interpretación de los blancos (si se quiere describir algun separador delimitador de datos fácilmente se puede determinar **IFS** , e incluir estos delimitadores), el shell inicialmente determina **IFS** e incluye un blanco, tabulador y carácteres que indican nuevas lineas.

MAIL Es el nombre de ruta del archivo donde el correo es depositado, si **MAIL** es determinado, entonces el shell verifica y ve si alguien a añadido correo a este archivo para hacer el anuncio de arribo de correo.

PATH

Especifica la ruta usada por el shell en la busqueda de comandos, esta variable es una lista ordenada de nombres de rutas de directorios separados por 2 puntos (:), el shell inicializa la variable **PATH** con la lista.

PATH=:bin:/usr/bin

: Donde un argumento nulo aparece frente a los primeros 2 puntos. Un argumento nulo en cualquier parte de la lista de la ruta de busqueda representa el directorio actual, en algunos sistemas una busqueda del directorio actual no es el modo de dafault y la variable **PATH** es inicializada preferentemente por:

PATH=/bin:/usr/bin

Si se desea que la busqueda de comandos se haga primero en el directorio bin, luego en el directorio /usr/bin y por último en el directorio actual la ruta sería:

PATH=/bin:/usr/bin::

Donde los 2 dos puntos (::) representan 2 puntos separados por un argumento nulo seguido por otros 2 puntos, de esta manera se nombra al directorio actual. Si se quiere poner un directorio personal de comandos (por ejemplo $HOME/bin) y que la busqueda empiece por el directorio de comandos y luego los otros 3 directorios (actual, /bin y /usr/bin) la ruta sería.

PATH=$HOME/bin::/bin:/usr/bin

La variable **PATH** es normalmente determinado en su archivo .profile

PS1
Esta variable especifica el prompt primario, si el shell es interactivo este prompt con el valor de **PS1** espera las entradas. El valor por default es **$** (signo de dolar) seguido por un espacio.

PS2 Esta variable especifica el prompt secundario, si el shell espera más entradas, cuando este encuentre una nueva línea a esta entrada, este prompt con el valor de PS2 espera dichas entradas, el valor por default es > (signo de mayor que) seguido por un espacio.

VARIABLES ESPECIALES PREDEFINIDAS

Algunas variables tienen un especial significado, las siguientes variables son determinadas solamente por el shell.

$#
Registra el número de argumentos pasados a el shell, pero no cuenta el nombre del procedimiento del shell, por ejemplo:

```
sh cmd a b c
```

$# obtiene el número de argumentos determinados por los parámetros posicionales, automaticamente $#=3 (los parametros a, b y c). Uno de los usos principales de esta variable es verificar el número de parámetros requeridos, ejemplo:

```
if test $# -lt 2
then
  echo " 2 o mas argumentos son requeridos "
  exit
fi
```

$? Contiene el número de salida del status (también se refiere al retorno de un codigo de salida o de un valor) del último comando ejecutado, este valor es una cadena décimal. La mayor parte de los comandos del sistema operativo retornan el cero que indica que su ejecucion fue lograda correctamente. El shell (sh) también retorna el valor actual de $? cuando este es terminado.

$$ Nos proporciona el número del proceso actual, porque los números de procesos son únicos, este valor es muchas veces usado para generar nombres de archivo temporales únicos. El sistema operativo no provee un mecánismo para la creación automática y borrado de archivos temporales. Un archivo existe hasta que este es explicitamente removido, los archivos temporales son generalmente indeseados ; el sistema operativo lo útiliza para muchas aplicaciones. Sin embargo, es necesario para crear archivos temporales con nombre único y que acasionalmente ocurren.

El siguiente ejemplo ilustra la práctica recomendada para crear archivos temporales con nombre único.

```
# usa los procesos actuales id
# para formar archivos unicos temporales
temp=/usr/tmp/$$
ls > $temp
# los comandos que estan aqui algunos de ellos usan $temp
rm $temp
```

$! El número del ultimo proceso corrido en background (&), este es una cadena que contiene de 1 a 5 dígitos.

$-** Es una cadena que consiste de nombres de banderas de ejecucion actuales, vueltas sobre el shell. Por ejemplo, **$- tiene el valor xv si estamos trazando la salida.

EL ESTADO DEL SHELL

Es determinado por los valores de los parametros posicionales, variables definidas por el usuario, de Ambiente, modos de ejecucion y trabajando en el directorio actual. El estado del shell puede ser alterado en varios caminos, cambiando el directorio actual de trabajo con el comando **cd**, determinado varias banderas y leyendo comandos desde el archivo especial .profile en su directorio de casa.

CAMBIANDO DIRECTORIOS

El comando **cd** cambia el directorio actual de trabajo a otro especificado como argumento de dicho comando, se puede poner el comando **cd** dentro de paréntesis para que ejecute en un subshell el cambio a un diferente directorio y ejecute algunos comandos fuera del shell de origen. Por ejemplo la siguiente secuencia de abajo copia el archivo /etc/passwd a /usr/marlon/passwd.

cp /ect/passwd /usr/marlon/paswwd

Esta otra secuencia primero se cambia al directorio /etc y luego copia el archivo passwd a el archivo /usr/marlon/passwd

cd /etc ; cp passwd /usr/marlon/passwd

EL ARCHIVO .PROFILE

El archivo llamado .profile es leido cada vez que se entra al programa login a traves del sistema operativo, este archivo es normalmente usado para ejecutar comandos solamente a un tiempo, para determinar algunas condiciones establecidas y para exportar las variables nombradas en este archivo a todos los shell que se ejecutaran más tarde. Solamente después los comandos son leídos y ejecutados desde el .profile, el shell lee los comandos desde la entrada estandar (usualmente la terminal).

INVOCANDO A EL SHELL

El shell es un comando y puede ser invocado en algunos caminos como cualquier otro comando.

sh proc[arg ...] Una nueva instancia del shell es explicitamente invocada para leer proc. Cualquier argumento puede ser manipulado.

sh -v proc[arg ...]
Esto es equivalente a poner **set -v** en el comienzo de proc. Esto puede ser usado en algun camino por la **-x**, **-e**, **-u** y **-n** banderas.

proc[arg ...]
Si proc es un archivo ejecutable y no es un programa compilado ejecutable, el efecto es similar a :

sh proc args

Una ventaja de esta forma que las variables son exportadas en el shell esto podrá ser exportado desde proc, cuando esta forma es usada (porque el shell solamente se bifurca para para leer comandos desde proc). Asi cualquier cambio hecho dentro de proc los valores de las variables exportadas son pasadas sobre los subsecuentes comandos invocados desde proc.

PASANDO ARGUMENTOS A PROCEDIMIENTOS DEL SHELL

Cuando una linea de comandos es explorada, cualquier secuencia de carácteres de la forma **$n** es remplazada por los enesimos argumentos del shell, contando el nombre del procedimiento del shell como $0. Esta notación permite hacer directamente referencia al nombre del procedimiento y a cualquiera de los 9 parámetros posicionales, se pueden procesar argumentos adicionales con el comando **shift** o el ciclo **for**.

El comando **shift** cambia los argumentos de la izquierda, el valor de $1 es desoprovechado, $2 remplaza a $1, $3 remplaza a $2 asi sucesivamente. El parametro opcional más alto llega ser considerado ($0 nunca es remplazado). Por ejemplo en el siguiente procedimiento del shell llamado rizo, el comando **echo** escribe los argumentos a la salida estandar.

```
# comando rizo
while test $# != 0
do
  echo $1 $2 $3 $4 $5 $6 $7 $8 $9
  shift
done
```

Si el procedimiento es invocado con rizo a b c se producirá una salida de la siguiente forma.

```
a b c
b c
c
```

La variable especial del shell $* causa una substitución de todos los parámetros posicionales excepto $0, asi la línea contenido el comando **echo** del programa rizo puede ser substituido por **echo $***

Estos 2 comandos echo no son equivalentes, el primero imprime hasta 9 parámetros posicionales y la variable del shell $* es más concisa y menos propensa a errores. Una aplicación obvia es pasar un nomero arbitrario de argumentos a un comando. Por ejemplo:

```
$  wc $*
```

Este comando cuenta las palabras en cada archivo nombrado (todos) en la linea de comando.

Esto es importante para entender la secuencia de acciones usadas por el shell en la busqueda y la subtitucion de los argumentos. El shell primero lee las entradas ya sea que esten en una nueva linea o esten separadas por punto y coma (;), luego hace un analisis gramátical de la entrada . Las variables son remplazadas por estos valores y los comandos de substitucion son (via comillas abiertas) procurados. La redireccion I/O de los argumentos son detectados, sobrepuestos y borrados desde la linea de comandos. Siguiendo, el shell explora el resultado de la linea de comando por el separador de campo interno, este es cualquier carácter especificado por **IFS** que rompe la linea de comandos dentro de argumentos distintos: argumentos nulos explicitos (especificados por "" o '') son retenidos, mientras los argumentos nulos implicitos resultantes desde la evaluación de variables, éstas son nulas o no son determinados para ser removidas. Entonces la generacion de nombres de archivos ocurren con todos los metacaracteres expandidos y el resultado en la linea de comando es entonces ejecutado por el shell.

Algunas veces las lineas de comandos son creados dentro de un procedimiento del shell, en este caso, algunas veces útil para tener el shell listo en la linea de comando, despues todas las substituciones

iniciales y expresiones tienen que ser ejecutadas.

El comando especial **eval** disponible para este propósito toma una linea de comando como este argumento y simplifica la linea, ejecutando la variable especificada o comandos substituidos.

Considere la siguiente situación (simplificada)

```
quienes = who
lineas = ' | wc -l'
eval $quienes $lineas
```

El siguiente segmento resulta de la ejecución de la linea de comando **eval**.

```
who | wc -l
```

La salida de **eval** no puede ser redireccionada, sin embargo, el uso de **eval** puede ser necesitada en una linea de comando y ser evaluado algunas veces.

ESTRUCTURAS DE CONTROL DEL SHELL

Diversos comandos del shell implementan una gran variedad de estructuras de control, útiles para determinar el control de flujo desde un procedimiento del shell a otro, antes de describirse estas estructuras, unos pocos terminos deben ser definidos.

Un simple comando es cualquier comando simple irreducible, especificado por el nombre de un archivo ejecutable, la redireccion de los argumentos pueden aparecer en una simple linea de comando y son pasados al shell y no al comando.

Los comandos de control del shell descritos en esta sección, en una secuencia de uno o más comandos separados por un pipe (|), la salida de cada comando (excepto el último) es conectado a la entrada estandar de el siguiente comando. Cada comando conectado por un pipe es corrido separadamente; el shell espera a que el último comando finalice. La salida de status de una secuencia de comandos conectados por pipes es diferente de cero si la salida del status de cualquiera de los primeros o últimos procesos en la secuencia de comandos es diferente de cero.

Una lista de comandos es una secuencia de uno o más (o secuencia de comandos conectados por pipes) separados por (;), un ampersand (&), un "y-si" simbolizado por (&&) o un "o-si" simbolizado por (||) y opcionalmente terminados por (;) o un (&).

Un (;) causa una ejecución secuencial de comandos conectados por pipes, esto significa que el shell espera, a que la secuencia finalice antes de leer la siguiente secuencia de comandos.

Un ampersand al final de una línea de comando causara una ejecución en background en forma asincrona y continuara ejecutandose esta línea hasta que el proceso termine o sea matado. Por ejemplo:

$ cc tablas.c &

Se puede seguir trabajando mientras el compilador del C se esta ejecutando en background. Una línea de comando es inmune a las interrupciones o salidas que se generen por la escritura **INTERRUP** o **QUIT** o la ejecución de **Ctrl-d.**

Los operadores **&&** y || causan ejecuciones condicionales de una secuencia de comandos, pero estos son de igual precedencia cuando las líneas de comandos son evaluadas (pero ambos operadores son de menor precedencia que el ampersand (&) y el pipe (|)) en la siguiente línea de comando.

comando1 || comando2

El comando1 es ejecutado y su salida es examinada , solamente si comando1 falla (si tiene una salida de status diferente de cero), entonces comando2 es ejecutado, una notación más clara sería :

```
if comando1 test $? != 0   # evaluacion del status
then
  comando2       # ejecucion de comando2 falla comando1
fi
```

El operador **&&** produce una prueba complementaria, por ejemplo :

comando1 && comando2

El segundo comando es ejecutado solamente si comando1 tiene exito (y tiene salida status cero). En la siguiente secuencia, cada comando es ejecutado en orden hasta que uno de ellos falle.

comando1 && comando2 && comando3 && . . . . && comandon

Un simple comando en un pipe conectado con una secuencia de comandos puede ser remplazado por una lista de comandos encerrados en parentesis () o entre llaves {}, la salida de todos los comandos que esten encerrados, es combinada dentro de un flujo que llega a ser la entrada del siguiente comando en la secuencia de comandos.

La siguiente línea formatea e imprime 2 documentos separados :

{ nroff -mm tésis.txt; nroff -mm sed.ing; } | lpr

Notese que es necesario el espacio despues de la llave izquierda y el punto y coma, debe aparecer antes de la llave derecha.

USO DE LA SENTENCIA IF

El shell provee una estructura de control condicional, la sentencia **if** y esta tiene la forma siguiente :

```
if lista_de_comandos1
then
   lista_de_comandos2
fi
```

Cuando lista_de_comandos1 es ejecutado y si el último comando de la lista su status es diferente de cero, entonces se ejecutará lista_de_comandos2. La palabra **fi** indica el fin del comando **if**

Para causar una determinada alternativa de ejecución de comandos cuando la salida del status es diferente de cero, se usa entonces la cláusula **else**, que puede ser dada con la siguiente estructura :

```
if lista_de_comandos1
then
   lista_de_comandos2
else
   lista_de_comandos3
fi
```

Múltiples pruebas pueden ser archivadas en un comando **if** usando la cláusula **elif**, aunque la sentencia **case** es mejor para numerosas pruebas, por ejemplo :

```
if test -f '$1'
then
  echo ' $1 es un archivo '
  lpr $1
elif test -d '$1'
  then
    echo ' $1 es un directorio '
    cd $1
  else
    echo ' $1 no es un archivo ni tampoco un directorio '
fi
```

La salida del status de la sentencia **if** es la salida del status del último comando ejecutado en cualquier cláusula **then** o la cláusula **else**. Si alguno de estos comandos no es ejecutado satisfactoriamente, **if** retorna un cero o la salida del status.

Una notación altenativa para el comando **test** es usar corchetes y encerrar esta expresión dentro de estos. Siguiendo el ejemplo previo.

```
if [ -f '$1' ]
then
  echo ' $1 es un archivo '
  lpr $1
elif [ -d '$1' ]
  then
    echo ' $1 es un directorio '
    cd $1
  else
    echo ' $1 no es un archivo ni tampoco un directorio '
fi
```

Debe notarse los espacios antes y despues de los corches ya que son escenciales en esta forma de sintáxis.

USANDO LA SENTENCIA CASE

Una multiple prueba condicional es prevista por el comando **case**, su formato básico de este comando es :

```
case cadena in
  patron ) lista_de_comando1
     ;;
  patron ) lista_de_comando2
     ;;
  ....
  patron ) lista_de_comandon
esac
```

El shell compara **cadena** con cada **patron** . si cadena = patron, entonces se ejecuta la lista_de_comando que corresponda. los dobles punto y coma (;;) sirver como un rompimiento fuera de **case** y es requerido despues de cada lista_de_comando excepto en el ultimo, además debe notarse que solo un patron es siempre comparado y que estas comparaciones son hechas en orden. si un asterisco * es el primer patron en la sentencia **case**, entonces los demas patrones no son revizados.

Más de un patrón puede ser asociado con una lista_de_comando dado y especificar los patrones alternados separados por una barra vertical (no es un pipe o conducto).

```
case $1 in
  *.c )              cc $1
   ;;
  *.h | *.sh )     : # los 2 puntos son arg nulos (no hacer nada)
   ;;
  * )                echo ' $1 es de un tipo desconocido '
   ;;
esac
```

En el ejemplo de arriba ninguna acción es llevada a cabo por el

segundo caso, porque el comando nulo (:) es especificado. El asterisco (*) es usado como un patron de default, ya que este hace juego con cualquier patrón.

La salida del status del comando **case** es la salida del status del último comando ejecutado, si ningun comando es ejecutado, entonces el comando **case** tiene una salida de estatus igual a cero.

CICLOS CONDICIONALES : WHILE Y UNTIL

Una sentencia **while** tiene la forma general siguiente :

```
while lista_de_comando1
do
  lista_de_comando2
done
```

Los comandos en lista_de_comando1 son ejecutados y si la salida del status de cada comando ejecutado es cero, entonces los comandos que estan en lista_de_comando2 son ejecutados. Esta secuencia es repetida por largo que sea la lista_de_comando1 y además que la salida del status de cada comando ejecutado sea cero.

Un ciclo puede ser ejecutado por largo que sea lista_de_comando1 y que la salida del status sea diferente de cero para el comando **until**.

Para el caso del comando **until**

```
until lista_de_comando1
do
  lista_de_comando2
done
```

Cualquier linea nueva en los ejemplos de arriba pueden ser remplazados por punto y coma (;). La salida del status de **while** o de **until** es la salida del status del último comando ejecutado en la lista_de_comando2, si ningún comando semejante es ejecutado, **while** o **until** tienen una salida de status igual a cero.

CICLOS PARA LISTAS : FOR

La ejecución de unas determinadas operaciones para cada archivo en un conjunto de archivos, o la ejecución de un comando, una vez por cada uno de los argumentos, para estos casos se usa el comando **for**. La sentencia **for** tiene la siguiente forma :

```
for variable in lista_de_palabras
do
  lista_de_comandos
done
```

Aqui la lista_de_palabras es una lista de cadenas separadas por espacios, la lista_de_comandos son ejecutados una vez por cada palabra que se encuentre en lista_de_palabras. Cada palabra en lista_de_palabras es puesta en variable, la lista_de_palabras es fijada despues y esta es evaluada a un tiempo.

Por ejemplo el siguiente ciclo causa que cada archivo fuente arch1.c arch2.c y arch3.c en el directorio actual sean comparados con los archivos de los mismos nombres pero en el directorio /usr/src/cmd/sh:

```
for C_ARCHIVOS in arch1 arch2 arch3
do
  diff ${C_ARCHIVOS}.c /usr/src/cmd/${C_ARCHIVOS}.c
done
```

Notese que la primera ocurrencia de C_ARCHIVOS inmediatamente después de la palabra **for** no esta precedida por el signo de dolar ($), despues el nombre de la variable es posicionado y no su valor.

Se puede omitir la palabra (**in** lista_de_palabra) de el comando **for**; esto causa la actual determinación de los parámentros posicionales sean usados en lugar de lista_de_palabras. Esto es útil cuando escribimos un comando y este ejecute algunos determinados comandos por cada número de argumentos desconocido.

Si creamos un archivo llamado eco2 con el siguiente contenido :

```
for palabra
do
   echo $palabra$palabra
done
```

Proporcionemos a este archivo los permisos de ejecución

```
$ chmod +x eco2
```

Y ahora escribimos lo siguiente:

```
$ eco2 ma pa bo fi yo no so ta
```

La salida de este comando será :

```
mama
papa
bobo
fifi
yoyo
nono
soso
tata
```

CONTROL DE CICLOS : BREAK Y CONTINUE

Se puede usar la sentencia **break** para que termine la ejecucion de un ciclo **while** o un ciclo **for**, ademas se puede usar tambien **continue** para efectuar la ejecución de la siguiente iteracción del ciclo. Estos comandos son efectivos solamente cuando ellos aparecen entre **do** y **done**

La sentencia **break** termina una ejecución del ciclo anterior mas próximo a el, es decir causa la salida hacia la palabra **done**. Para salirse de **n** niveles de ciclos, use la forma **break n**.

La sentencia **continue** causa que la ejecucion continue a la siguiente interaccion de los comandos **while, until** o **for**, además se puede especificar **continue n** para que se continue al eneavo ciclo.

```
# este es un procedimiento interactivo
# los comandos break y continue son utilizados
# para asignar el control de datos de entrada al usuario

while true          # ciclo perpetuo
do
  echo -n " inserte un dato de entrada    break o continue : "
  read respuesta
  case "$respuesta" in
    "done"       ) break      ;;  # rompimiento del ciclo  while
    "continue"   ) continue   ;;  # continuacion del ciclo while
    *            )            ;;
  esac
done
```

FINALIZANDO UN PROCEDIMIENTO DEL SHELL : FIN DE ARCHIVO Y SALIR

Cuando el shell alcanza el fin de archivo en un procedimiento del shell, este termina la ejecución, retornando la salida del status de el último comando ejecutado anteriormente por el fin de archivo el primer nivel del shell es terminado escribiendo **Ctrl-d**, el cual es equivalente a terminar la sesión.

El comando **exit** simplemente lee el fin de archivo y regresa al ambiente con la salida de status de cualquiera de estos argumentos. Asi un procedimiento puede ser terminado si **exit 0** es puesto al final del archivo.

AGRUPACION DE COMANDOS : PARENTESIS Y LLAVES

Estos son 2 métodos para agrupar comandos en el shell: los paréntesis () y las laves {} . Los paréntesis hacen que el shell cree un subshell que lea los comandos encerrados dentro de los paréntesis, pero los paréntesis derecho e izquierdo son reconocidos donde quiera que aparezcan en la línea de comando. Estos pueden aparecer como paréntesis literales solamente cuando estan encerrados con las comillas dobles . Por ejemplo si se escribe :

```
$ garble(stuff)
```

el shell imprime un mensaje de error, para evitar esto haga lo siguiente:

```
$ garble"("stuff")"
$ "garble(stuff)"
```

De esta forma son interpretados correctamente. Otros mecánismos de las comillas son discutidos en la sección de "Mecanismos de los Comillas".

La agrupación de comandos es útil cuando se quiere ejecutar operaciones pero que no afecten los valores de las variables del shell actual, esto tambien asigna cambios temporales en el directorio de trabajo y ejecuta los comandos en el nuevo directorio teniendo que regresar al directorio actual.

El ámbiente actual es pasado al subshell, así como las variables del shell actual son exportadas tambien al subshell, así :

```
DIRECTORIO_ACTUAL = `pwd`
cd /usr/docs/otro_dir
nohup nroff doc.n | lpr&
cd $DIRECTORIO_ACTUAL
```

y

```
( cd /usr/docs/otro_dir ; nohup nroff doc.n | lpr& )
```

Una copia de /usr/docs/otro_dir/docs.n es mandada a la impresora e inmediatamente regresamos al directorio actual de trabajo a través del comando **cd**. El comando **nroff** esta disponible en el sistema operativo como procesador de texto. Sin embargo el segundo ejemplo automaticamente regresa al directorio original de trabajo ya que este se ejecutó en un subshell (no es necesario el comando **cd**), además los espacios o las nuevas líneas son asignados pero no necesarios. Cuando se introduce una línea de comando desde la terminal el shell despliega el **prompt** con el valor de la variable del shell PS2 si el paréntesis izquierdo no cupo en la primera línea.

Las llaves { y } pueden tambien agrupar comandos al mismo tiempo. Pero la llave derecha es reconocida solamente si ésta aparece como la primera palabra de un comando. La llave derecha puede ser seguida por una nueva línea (en el caso de que el shell necesite más de una línea de entrada).

De manera diferente de los parentesis, las llaves no crean un subshell, sino que los comandos dentro de las llaves son simplemente leidos por el shell. Las llaves son convenientes cuando se quiere usar la salida (secuencial) de diversos comandos como la entrada da un comando.
La salida del status de un determinado grupo de comandos dentro de los paréntesis o las llaves, es la salida del status de el último comando ejecutado.

REDIRECCIONAMIENTO DE ENTRADA/SALIDA (I/O) Y CONTROL DE COMANDOS

El shell normalmente no es único y crea un nuevo shell, cuando este reconoce los comandos de control (otros parentesis) descritos en la sección previa. Sin embargo, cada comando en una secuencia de comandos conectados por pipes es corrido como un proceso separado en orden a la dirección de entrada o salida de cada comando. También, cuando la redirección de entrada o salida es especificada explicitamente al comando de control, un proceso separado es producido para ejecutar este comando. Así, cuando los comandos **if, while, until, case** y **for** son usados en una secuencia de comandos conectados por pipes, el shell se ramifica y un subshell corre los comandos de control. Esto tiene 2 implicaciones :

1.- Cualquier cambio hecho a las variables dentro de el comando de control no son efectivos una vez que el comando de control finaliza (esto es similar a el efecto de usar parentesis para un grupo de comandos)

2.- Los comandos de control corren ligeramente despacio cuando son redireccionados, porque el shell crea una parte adicional (shubshell) para los comandos de control.

TRANSFERENCIA DE UN ARCHIVO Y REGRESO : EL COMANDO PUNTO (.)

Una linea de comando de la forma

. proc

Causa que el shell lea los comandos desde **proc**, mientras que por fuera se esta produciendo un nuevo proceso. Los cambios en las variables en **proc** son efectuados despues de que el comando (.) finalice

Este es un buen camino para recoger un número de la variable del shell inicializada dentro de un archivo. Un uso común de este comando es la reinicialización del primer shell que se efectua leyendo el archivo **.profile** de la forma siguiente :

. .profile

COMANDOS DE APOYO Y CARACTERISTICAS

Los procedimientos del shell pueden hacer uso de cualquier comando del sistema operativo, los comandos descritos en esta sección, describen como son usados frecuentemente en procedimientos del shell o cuando tienen que ser explicitamente designados para varios usos.

EVALUACION CONDICIONAL : TEST

El comando **test** evalua las expresiones especificadas en sus argumentos y regresa una salida de status igual o cero si la expresión es cierta, si la expresión es falsa, **test** regresa una salida de status diferente de cero. El comando **test** tambien regresa una salida de status diferente de cero si no se le proporcionan argumentos. Con frecuencia esto es conveniente para usar el comando **test** como el primer comando en la lista que sigue a un **if** o a un **while**. Las variables del shell usadas en **test** deben ser encerradas en dobles comillas si cualquiera de estos cambios empiezan con nulos o no son determinados.

Los corchetes pueden ser usados como un alias de test de la siguiente manera :

[expresion]   tiene el mismo efecto que   **test** expresión

Notense los espacios antes y despues de expresión ya que estos son escenciales.

El siguiente ejemplo es una lista parcial de las opciones que pueden ser usadas para la construcción de expresiones condicionales :

-r archivo
Cierto si el archivo nombrado es leible por el usuario

-w archivo
Cierto si el archivo nombrado existe y se puede escribir sobre él, por el usuario.

-x archivo
Cierto si el archivo nombrado existe y es ejecutable por el usuario

-s archivo
Cierto si el archivo nombrado existe y tiene un tamaño mayor que 0

-d archivo
Cierto si el archivo nombrado es un directorio

-f archivo
Cierto si el archivo nombrado es un archivo ordinario

-z s1
Cierto si el largo de cadena de s1 es mayor que cero

-c archivo
Cierto si el archivo existe y es un caracter especial de archivo

-b archivo
Cierto si el archivo existe y es un bloque especial de archivo

-u archivo
Cierto si el archivo existe y este determina el bit ID del usuario

-g archivo
Cierto si el archivo existe y este determina el bit ID del grupo

-k archivo
Cierto si el archivo existe y este bit dificil es determinado

-p archivo
Cierto si el archivo existe y es un nombrado pipe (fifo)

-ns1
Cierto si el largo de cadena de s1 es diferente de cero

-t [fildes]
Cierto si el archivo abierto cuyo numero de archivo descriptor es fildes es asociado con un dispositivo de una terminal. Si fildes no es especificado, el archivo descriptor 1 es usado por default.

s1 = s2
Cierto si la cadena s1 y s2 son identicos

s1 != s2
Cierto si la cadena s1 y s2 no son identicos

s1
Cierto si s1 no es una cadena nula

n1 -nq n2
Si los enteros n1 y n2 son algebraicamente iguales

n1 -gt n2
Si el entero n1 es mayor que n2

n1 -ge n2
Si el entero n1 es mayor o igual a n2

n1 -lt n2
Si el entero n1 es menor que n2

n1 -le n2
Si el entero n1 es menor o igual n2

Las opciones pueden ser combinados con los siguientes operadores :

!	Operador de negacion unaria
-a	Operador de logica binaria AND

-o Operador de lógica binaria OR; este tiene menor precedencia que el operador lógico AND (-a)

(expr) Paréntesis de agrupamiento; este debe ser escapado para remover estas significancias al shell. En la ausencia de los paréntesis, la evaluación se procede de izquierda a derecha.

Notese que todas las operaciones, operadores, nombres de archivo etc, son argumentos separados de **test**.

USO DEL COMANDO ECHO

El comando **echo** repite sus argumentos

su sintáxis es

echo [-e] [-n] [-u] [--] [arg]

El comando **echo** repite sus argumentos sobre la terminal. Los argumentos deben ser separados por blancos y terminar por una nueva línea.

El comando **echo** es usado cuando se escriben procedimientos del shell para marcar cada paso en los procedimientos, además produce diagnosticos en los programas del shell y escribe datos constantes sobre pipes. Para enviar diagnosticos al archivo de error estandar escribase.

' echo .... 1>&2 '

opciones :

-e Imprime los argumentos sobre la salida estandar

-n Imprime la línea por fuera de una línea nueva

-u Imprime I/O sin usar buffer

-- imprime arg exactamente ___

El comando **echo** usa las convenciones de secuencias de escape de C. Las siguientes secuencias de escape necesitan ser encerradas en comillas simples cerradas para que el shell las interprete en forma correcta :

'\b' Backspace (ignorado si el último carácter esta en una cadena)

'\c' Imprime la linea fuera de una linea nueva

'\f' Alimentación de una nueva forma

'\n' Nueva linea

'\r' Retorno de carro

'\t' Tabulador

'\\' Interpreta literalmente \

'\n' Los caracteres de 8 bit cuyo código ASCII es el 1-, 2- o 3 digitos de número octal n, el cual debe empezar con cero.

Nota:

Si se usan los comando **sh echo** , se debe especificar el nombre de ruta completo :

/bin/echo

Pero si se usan los comandos **csh echo** ver el comando **csh**(CP)

EVALUACION DE EXPRESIONES : EXPR

Los argumentos son tomados como expresiones, después de evaluarlos, los resultados son escritos sobre la salida estandar. Los terminos de una expresión deben ser separados por blancos y los caracteres especiales del shell deben ser escapados. Cuando el cero es regresado, se esta indicando un valor cero antes que la cadena nula. Las cadenas contienen blancos u otros caracteres especiales y deben ser encerrados entre comillas. Los argumentos valuados como enteros pueden ser precedidos por el signo menos. Internamente los enteros son tratados como 32 bits y como complemento a 2.

Las expresiones deben ser encerrados por el shell, desde entonces muchos de los carácteres especiales tienen especial significado en el shell, así como tambien tienen especial significado en el comando expr La lista es en un orden de precedencia creciente, con igual precedencia para los operadores agrupados con llaves { }.

expresion1 | expresion2 retorna expresion1 si ninguna de las 2 expresiones es nulo o cero. Otro caso, retorna expresion2.

expresion1 & expresion2 retorna expresion1 si ninguna de las 2 expresiones es nulo o cero. Otro caso regresa un cero.

expresion1 { =, >, >=, <, <=, != } expresion2 regresa el resultado de una comparacion de un entero, si ambos argumentos son enteros. Otro caso, regresa el resultado de una comparación léxica.

expresion1 { +, - } expresion2 los argumentos enteros son evaluados con adición ó sustracción.

expresion1 { *, /, % } expresion2 los argumentos enteros son evaluados con multiplicacion, division o módulo.

expresion1 : expresion2 el operador : compara el primer argumento con el segundo. el cual debe ser una expresión regular, la sintáxis de una expresión regular es alguna expresión del comando ed(C), excepto que todos los patrones son asegurados (empezando con ^) y por lo tanto el símbolo carat (^) no es un caracter especual en este contexto. (notese que en el shell, el carat tiene el mismo significado especial que el símbolo pipe (|).) Normalmente el operador : regresa el número de caracteres que hicierón juego con el otro patrón (cero o omisión). Alternativamente, el símbolo \(...\) puede ser usado para retornar una porción de el primer argumento **Ejemplos :**

`a=`**`expr`**` $a + 1`` Añade 1 a la variable del shell a

`'`**`expr`**` $a : .*/\(.*\) | $a'` Para $a igual a "/usr/marlon/archivo" a "archivo"

Regresa el último segmento de un nombre de ruta o pathname. Se debe tener cuidado con el símbolo / solo, como un argumento: el comando **expr** puede interpretar este símbolo como el operador de división.

`expr`` $VAR : '.*'` Regresa el número de caracteres en $VAR

Diagnósticos

El comando **expr** regresa los siguientes valores de salida de status

0 Si la expresión no es nulo o cero

1 Si la expresión es nulo o cero

2 Para expresiones invalidas

Otros diagnósticos incluidos son :

syntax error Para errores de operadores y operandos

nonnumeric argument Si una operación aritmética es intentado como una cadena.

Notas

Después de que los argumentos son procesados por el shell, **expr** no puede decir la diferencia entre operador y operando excepto por el valor. Si $a es igual al símbolo =, el comando :

expr $a = = se mira como **expr = = =**

Así los argumentos son pasados a **expr** (y todos son tomados como el operador =). El siguiente ejemplo permite comparar signos =

expr X$a = X=

SALIDA DE STATUS : TRUE Y FALSE

Los comandos **true** y **false** ejecutan las funciones de salidas del status con cero y diferente de cero respectivamente. Estos comandos son frecuentemente usados para ciclos condicionales. Por ejemplo :

```
while true
do
  echo  ciclo para siempre
done
```

Los argumentos del comando **echo** seran desplegados continuamente en la terminal, para finalizar el ciclo se interrumpe con **Break/Del**

ENTRADA DE DOCUMENTOS EN LINEA

Cuando el shell ve una linea de comando de la forma :

command << string

Donde string es cualquier cadena, esto toma las subsecuentes lineas como la entrada estandar de command mientras una linea sea leida solamente de string (si se añade un - a el símbolo de redirección de entrada ..., los espacios y tabuladores son borrados desde cada linea de la entrada del documento antes de que el shell los pase a la linea de command).

Shell crea archivos temporales conteniendo la entrada del documento y ejecuta las variables y comandos de sustitucion sobre este contenido antes de pasar estos al comando. Los juegos de patrones sobre nombres de archivos es ejecutado sobre los argumentos de las lineas de comandos en las sustituciones de comandos. En orden para prohibir todas las sustituciones, se puede encerrar cualquier carácter de string :

command << \string

La caracteristica de la entrada en linea de documentos es especialmente usada por pequeños cantidades de datos de entrada, donde este es más conveniente para poner el dato en el procedimiento del shell que a quedarse este en un archivo separado. Por ejemplo:

```
cat <<- xx

Este mensaje es impreso sobre la terminal con la caracteristica de
que los espacios y tabuladores son removidos.

xx
```

Esta característica de entrada de documentos en linea es más usada en procedimientos del shell. Notese que la entrada en linea de documentos puede no aparecer dentro con acentos gravados.

REDIRECCION DE ENTRADA/SALIDA USANDO ARCHIVOS DESCRIPTORES

Nosotros tenemos establecido, que un comando ocasionalmente direcciona la salida, algún archivo asociado con un archivo descriptor que pueden ser 1 o 2. El shell provee estos propios mecanismos para crear una entrada de archivo asociada con un particular archivo descriptor, si se escribe :

```
fd1>&fd2
```

Donde **fd1** y **fd2** son archivos descriptores validos, uno puede direccionar la salida y esta puede ser normalmente asociado con el archivo descriptor **fd1** a el archivo asociado con **fd2**. El valor por default por **fd1** y **fd2** es **1**. Si en un tiempo de ejecucion, el archivo no es asociado con **fd2**. Entonces la redirección es vacia.

El uso más común de este mecánismo es el de direccionar la salida del error estandar a el mismo archivo como la salida estandar, esto es

```
comando 2>&1
```

Si se quiere que la redirección de la salida estandar y la salida del error estandar sea a algún archivo se escribiría :

```
comando 1>archivo 2>&1
```

El orden de la sentencia anterior es significante : Primero, el archivo descriptor 1 es asociado con **archivo**; entonces el archivo descriptor 2 es asociado con el mismo **archivo** como es actualmente asociado con el archivo descriptor **1**.

Si el orden de las redirecciones es en forma inversa, la salida del error estandar es dirigida a la terminal y la salida estandar es dirigida a **archivo**, porque a un tiempo de la redirección de la salida

del error estandar, el archivo descriptor 1 aun tiene que ser asociado con la terminal.

Este mecanismo puede también ser generalizado para redireccionar la entrada estandar, de la siguiente manera :

fda<&fdb

Causa que cualquiera de los 2 archivos descriptores fda y fdb sean asociados con el mismo archivo de entrada. Si fda y fdb no son especificados, entonces el archivo descriptor 0 es asumido. Cada redirección de entrada es usada por un comando que usa 2 o más entradas

SUBSTITUCIONES CONDICIONALES

Normalmente el shell remplaza las ocurrencias de $variables por la cadena de valores asignada a variable. Sin embargo, una notación especial, asigna substituciones condicionales dependiendo de si la variable es determinada o nula. Por definición, una variable es determinada si esta tiene que ser asignada a un valor. El valor de una variable puede ser una cadena nula, la cual puede ser asignada a una variable por varios caminos :

```
variable=
variable=""
variable=''
```

Los ejemplos anteriores asignan nulos de diversas maneras a las variables del shell.

El siguiente ejemplo determina el primero y segundo parámetro posicional como valores nulos.

```
set '' ""
```

Las siguientes expresiones condicionales dependen de si una variable es determinada y no es nula, además de que el significado de las llaves usadas para estas expresiones tienen un significado diferente cuando son usados en agrupamiento de comandos del shell, y los parámetros se refieren ya sea aun digito o a un nombre de variable

${variable:-cadena}
Si **variable** es determinada y no es nula, entonces substituye el valor de $variable en lugar de esta expresión. Otro caso, remplaza la expresión con **cadena**. Notese que el valor de variable no es cambiado por la evaluación de esta expresión.

${variable:=cadena}
Si **variable** es determinada y no es nula, entonces substituye el valor de $variable en lugar de esta expresión. Otro caso, se determina **variable** a **cadena**, y entonces subtituye el valor de $variable en lugar de esta expresión. Los parámetros posicionales tal vez no sean asignados los valores en éste modo.

${variable:?cadena} Si **variable** es determinado y no es nulo, entonces substituye el valor de variable para expresión. Otro caso,

imprime un mensaje de la forma.

variable:cadena

y sale de el shell actual. (Si el shell es principal entonces la salida no es llevada a cabo). Si **cadena** es omitida en esta forma entonces el mensaje sería:

variable: parameter null or not set

${variable:+cadena} Si **variable** es determinada y no es nulo, entonces substituye **cadena** por esta expresión. Otro caso, substituye la cadena nulo. Notese que el valor de **variable** no es alterada por la evaluación de esta expresión.

Estas expresiones también pueden ser usadas por fuera los 2 puntos. en esta variación, el shell no reviza si la variable es o no nulo; este solamente reviza si la variable tiene alguna vez esta forma determinada.

Los siguientes ejemplos ilustran el uso de esta fácilidad :

1. Este ejemplo ejecuta una asignación explícita a la variable PATH.

"PATH"=${PATH:-':/bin:/usr/bin'}

Lo anterior dice : Si PATH es determinado y no es nulo, entonces PATH se conserva con el valor actual; Otro caso, se determina el valor de PATH con la cadena cadena ':/bin:/usr/bin'

2. Este ejemplo automáticamente asigna a la variable HOME un valor

cd ${HOME:='/usr/gas'}

Si HOME es determinado y no es nulo, entonces se realiza el cambio de directorio al valor de $HOME; otro caso, se determina HOME con el valor '/usr/gas' y luego se realiza el cambio de directorio.

Invocación de Banderas

Los siguientes banderas pueden ser especificados sobre la línea de comandos del shell.

-i Si esta bandera es especificado, o si la entrada y salida del shell son ambos anexados a la terminal, el shell es interactivo. Por lo tanto un shell tiene.

interrupción (señal 2), terminar (señal 15), salir (señal 3)

que son las señales atrapadas e ignoradas

-s Si esta bandera es especificada o si no existe redireccionamiento de entrada/salida de los argumentos, el shell lee los comandos desde la entrada estandar. La salida del shell es escrita

a el archivo descriptor 2.

-c Cuando esta bandera es puesta, el shell lee los comandos desde la primera cadena seguida de la bandera, y los argumentos sobrantes son ignorados. Se usan dobles comillas para encerrar una cadena multipalabra en orden para asignarla a una variable de sustitución.

EFECTIVA Y EFICIENTE PROGRAMACION DEL SHELL

Esta sección muestra estrategias para escribir eficientes procedimientos del shell, que no sean rechazados estos recursos en los propósitos realizados. La razón primaria para seleccionar un procedimiento del shell que ejecute una función específica, es el de dejar un resultado deseado con un mínimo costo humano. En los procedimientos del shell a elaborarse se enfatiza en la simplicidad, claridad y legibilidad de los programas, pero la eficiencia también puede ser obtenida por el conocimiento de unas pocas estrategias. En muchos casos, una efectiva redesignación de un procedimiento existente, improvisa esta eficiencia pero reduciendo su tamaño y teniendo en cuenta su comprensibilidad. En cualquier caso debemos contar con plan que optimice un procedimiento del shell si este es demasiado lento o si consume muchos recursos del sistema.

Los procedimentos del shell son sometidos a algunos ciclos interactivos como otros programas (escribir alguna cantidad de códigos y optimizar solamente algunas partes importantes). El usuario debe de familiarizarse con el comando **time**, el cual puede ser usado para tomar el tiempo a los procedimientos del shell. Este comando es fuertemente recomendado ya que la intuicion humanana del tiempo es notoriamente incierto cuando es usado para estimar el tiempo de ejecución de los programas.

NUMERO DE PROCESOS GENERADOS

Cuando muchos comandos pequeños son ejecutados, el actual tiempo de ejecucion de los comandos pueden ser dominados por el superior de los procesos creados. Los procedimientos incurren en significantes ascensos de tales superiores, estos son ejecutados por muchos ciclos y estos generán secuencias de comandos que son interpretados por otro shell.

Si la eficiencia es parte de la programacion, es importante que se conozcan los comandos actuales que estan en el shell y cuales no lo estan.

A continuación se muestra una lista alfábetica de los comandos del shell.

break	**exec**	**newgrp**	**best**	**wait**
case	**exit**	**read**	**times**	**while**
cd	**export**	**readonly**	**trap**	**.**
continue	**for**	**set**	**unmask**	**:**
eval	**if**	**shift**	**until**	**{}**

Los paréntesis (), son usados dentro del shell, pero los comandos encerrados dentro de ellos son ejecutados como procesos hijos del shell actual. La mayor parte de los procedimientos observados, el número de procesos creados (no necesariamente simultaneos) pueden ser descritos por.

procesos = (K * n) + c

donde K y c son constantes, y n puede ser el número de argumentos del procedimiento, el número de líneas de algun archivo de entrada, el número de entradas de algún directorio o alguna otra cantidad. Eficientes improvisaciones son más comunmente ganados por la reducción del valor de K, algunas veces a cero.

Como un ejemplo mostraremos un procedimiento llamado **split**

```
split
trap 'rm temp$$; trap 0; exit' 0 1 2 3 15
start1=0 start2=0
b='[A-Za-z]'
cat > temp$$
  # leer stdin dentro de el archivo temp
  # salvar la longuitud original de $1. $2
if test -s "$1"
then
  strat1='wc -l < $1'
fi
if test -s "$2"
then
  start2='wc -l < $2'
fi
grep "$b" temp$$ >> $1
  # lineas con letras dentro de $1
grep -v "$b" temp$$ | grep '[0-9]' >> $2
  #lineas con numeros solamente dentro de $2
total="`wc -l < temp$$`"
end1="`wc -l < $1`"
end2="`wc -l < $2`"
lost="`expr $total - \($end1 - $start1\) \ -\($end2 - $start2\)`"
echo "$total leer, $lost tirar lejos"
```

Por cada iteracción de el ciclo, este es un **expr** adicional, por **echo** o por cualquier otro **expr**. Un adicional **echo** es ejecutado en la última línea. Si n es el número de líneas de la entrada, el número de procesos es:

2 * n + 1

Algunos tipos de procedimientos no son escritos usando el shell, por ejemplo, si uno o más procesos son generados por cada carácter en un archivo, este es una buena indicación de que el procedimiento puede estar escrito en **C**. Los procedimientos del shell no son usados para examinar o construir archivos con carácter a un tiempo.

NUMERO DE BYTES DE DATOS ACCESADOS

Esta parte es útil para considerar cualquier acción que reduzca el número de bytes leidos o escritos, esto puede ser importante para procedimientos en los cuales el tiempo es agotado pasando datos alrededor para unos pocos procesos, antes que se produzcan números largos de pequeños procesos. Algunos filtros acortan estas salidas, otros usualmente incrementan este. Estos caminos dan una forma de poner un acortamiento, primero cuando el orden es irrelevante. Por ejemplo :

```
sort archivo | grep patron
```

La instruccion anterior ordenara **archivo** y después buscara **patron** pero el ejemplo siguiente es mas eficiente, ya que desde la entrada clasifica, con esto se agiliza el proceso.

```
grep patron archivo | sort
```

REDUCIENDO LA BUSQUEDA DE DIRECTORIOS

La busqueda de directorios puede consumir un gran cantidad de tiempo, especialmente en las aplicaciones donde utiliza estructuras de directorios profundos y largos pathnames. Juiciosamente usa el comando **cd** (cambio de directorio), que puede ayudarle en la busqueda de largos pathnames (nombres de ruta) y así reducir el numero de directorios buscados. Como ejemplo si se ejecuta la siguiente linea de comandos.

```
ls -l /usr/bin/* > /dev/null
```

Se esta especificando que se listen todos los archivos que esten en /usr/bin

Pero si reescribimos la linea de comando anterior

```
cd /usr/bin; ls -l * > /dev/null
```

Primero se hara el cambio de directorio a /usr/bin y despues se enlistaran todos los archivos contenidos en este directorio, así la ejecución de la linea de comando es mas rápida ya que el camino de busqueda ha sido acortado.

ORDEN DE BUSQUEDA DE LOS DIRECTORIOS Y LA VARIABLE PATH

La variable path es un mecánismo conveniente para asignar la organización y distribucion de procedimientos. Sin embargo, este puede ser usado en un modo sensible o el resultado puede ser un gran incremento de recursos del sistema.

Los procedimientos de busqueda de un determinado comando, es leyendo cada directorio incluido en el pathname de la variable actual PATH. Como un ejemplo considere el efecto de invocar al comando nrroff (/usr/bin/nroff), cuando el valor de PATH es ':/bin:/usr/bin', la secuencia de directorios leidos es :

```
.
/
/bin
/
/usr
/usr/bin
```

Esto es un total de 6 directorios. Una larga lista asignada en la variable PATH ocasiona un incremento significativo en la secuencia de directorios leídos.

La vasta variedad de comandos de ejecución son los comandos encontrados en el directorio /bin y un número menor en /usr/bin. El descuido de la organización de la variable PATH puede ocasionar una innecesaria busqueda.

Los siguientes 4 ejemplos son ordenados desde el peor hasta el mejor con respecto a la eficiencia de busqueda de comandos.

```
:/usr/john/bin:/usr/localbin:/bin:/usr/bin
:/bin:/usr/john/bin:/usr/localbin:/usr/bin
:/bin:/usr/bin:/usr/john/bin:/usr/localbin
/bin::/usr/bin:/usr/jchn/bin:/usr/localbin
```

La primer lista de busqueda puede ser evitada, los otros son aceptables y el cambio entre estos es dictado considerando el cambio en la determinación de los comandos guardados en /bin y /usr/bin.

CAPITULO 4

EXPRESIONES REGULARES

Antecedentes

Una expresión regular define un conjunto de una o más cadenas de caráteres.

Varios de los programas de utilidad UNIX, incluyendo ed, vi, grep y sed, emplean expresiones regualares para localizar y reemplazar cadenas.

EXPRESIONES REGULARES

Una cadena de caracteres sencilla es una expresión regular que define una cadena de caracteres: ella misma. Una expresión regular más compleja utiliza letras, números y caracteres especiales para definir muchas cadenas de caracteres diferentes. Se dice que una expresión regular equivale a cualquier cadena que define.

CARACTERES

En la forma en que se utiliza en este apéndice, un carácter es cualquier caracter excepto un *NewLine*. La mayor parte de los caracteres se representan a sí mismos dentro de una expresión regular. Un carácter especial es uno que no se representa a sí mismo. Si se necesita utilizar un caracter especial para que se presente a sí mismo, consultese la sección de este apendice sobre 'Marcado de caracteres especiales.

Delimitadores

Un carácter, llamado delimitador, por lo general marca el principio y el final de una expresión regular. El delimitador es siempre un carácter especial para la expresión regular que delimita (es decir, no se representa a sí mismo, sino que marca el principio y el final de la expresión). Puede usarse cualquier carácter como delimitador, siempre y cuando utilice el mismo en ambos extremos de la expresión regular. Para simplificar el trabajo, todas las expresiones regulares de este apéndice llevan una barra diagonal como delimitador.

En algunos casos no ambiguos, no se requiere el segundo delimitador. En general puede omitirse el segundo delimitador va a ir seguido de **RETURN**.

CADENAS SENCILLAS

La expresión regular básica es una cadena sencilla que no contiene caracteres especiales, excepto los delimitadores. Una cadena sencilla se corresponde sólo a si misma.

Expresión regular	Significado	Ejemplos
/osada/	equivale a osada	osada,posada,rosada
/jueves/	equivale a jueves	jueves
/o no /	equivale a o no	o no, como no

CARACTERES ESPECIALES

Dentro de una expresión regular pueden utilizarse caracteres especiales para lograr que ésta equivalga a más de una cadena. Una expresión regular que incluye un carácter especial siempre concuerda con la cadena más larga posible que comience lo más lejos posible hacia el principio de la línea.

Punto

Un punto equivale a cualquier carácter.

Expresión regular	Significado	Ejemplos
/.pal/	equivale a todas las cadenas que contenga un espacio seguido por cualquier carácter seguido por **pal**	al palacio la palabra un palito
/.ind/	con cualquier carácter que que preceda a **ind**	lindo,rindo el indio

Corchetes

Los corchetes ([]) definen una clase de caracteres que corresponde a cualquier carácter sencillo dentro de los corchetes. Si el primer carácter que sigue al corchete izquierdo es un caract (^), los corchetes definen una clase de caracteres que equivale a cualquier carácter sencillo que no se encuentre dentro de los corchetes. Puede utilizarse un guión para indicar una clase de caracteres. Dentro de una definición de clase de caracteres, las barras diagonales invertidas, los asteriscos y los signos monetarios (se describen más adelante) sólo puede aparecer como el primer carácter después del corchete izquierdo, y el símbolo ^ sólo es especial si es el primer carácter después del corchete izquierdo.

Expresión regular	Significado	Ejemplos
/[bB]uen/	define la clase de caracteres que contiene **b** y **B**; equivale a un miembro de la clase de carácter seguido por **uen**	bueno, Bueno, buenos
/t[aeiou].d/	equivale a **t** seguida por una vocal minúscula, cualquier carácter y una **d**	tardio, tilde tunda, tordo tendrá

/número[6-9]/	equivale a un número seguido por un espacio y un miembro de la clase de caracteres	número 6 número 8 número 9
/[^a-zA-Z]/	equivale a cualquier carácter que no sea una letra.	1, 7, @, .,)

Asterisco

Un asterisco puede seguir a una expresión regular para representar cero o más ocurrencias de un equivalente de la expresión regular que lo preceda. La expresión regular puede incluir cualquiera de los caracteres especiales denidos con anterioridad (.[^ -]). Un asterisco despues de un punto corresponde a cualquier cadena de caracteres. (Un punto equivale a cualquier caracter, y un asterisco, a cero o más ocurrencias de la expresión regular precedente.) Una definición de la clase de caracteres seguida de un asterisco iguala a cualquier cadena de caracteres miembros de la clase de caracteres.

Expresión regular	Significado	Ejemplos
/ab*c/	equivale a **a** seguido por cero o mas letras **b** seguido (s) por una **c**	ac, abc, abbc, abbbc
/ab.*/	equivale a **ab** seguido por cero o mas caracteres seguido(s) por **c**	abc, abxc, ab45c, ab 756.435 x cat
/p.*ende/	equivale a **p** seguido por cero o mas caracteres seguido(s) por **ende**	prende, pende por ende, pretende
/[a-zA-Z]*	equivale a cualquier cadena compuesta solo por letras y espacios	cualquier cadena sin numeros ni signos de puntuacion
/(.*)/	equivale a la cadena mas larga posible entre (y)	(esto) y (aquello)
/([^)]*)/	equivale a la cadena mas corta posible que empiece con (y termine con)	(esto), (esto y aquello)

SIGNOS ^ Y $

Una expresion regular que comienza con un ^ solo puede equivaler a una cadena al principio del archivo. De forma similar, un signo monetario ($ al final de una expresion regular) es igual al final de la linea

Expresión regular	Significado	Ejemplos
/^T/	equivale a una T al principio de una linea	Todos tienen ... Tierra, mar ...
/^+[0-9]/	equivale a un signo mas seguido de un numero al principio de una linea	+545.72 +759 mantenga ..
/:$/	equivale a un signo de 2 puntos que termina una linea	... continua: ... esto es:

MARCADO DE CARACTERES ESPECIALES

Puede marcarse cualquier caracter especial (que no sea un digito ni un parentesis) precediendolo con una diagonal invertida. Cuando se marca un caracter especial, este se representa a si mismo

Expresión regular	Significado	Ejemplos
/fin\./	equivale a todas las cadenas que contengan fin seguido de un punto	El fin., afin., fin. El correo ...
/\\/	equivale a una sola diagonal invertida	\
/*/	equivale a un asterisco	*
/\[5\]/	equivale a la cadena [5]	[5]
/y\/o/	equivale a y/o	y/o

REGLAS

Las reglas siguientes rigen la aplicacion de expreciones regulares

LA EQUIVALENCIA MAS LARGA POSIBLE

Como se mencionó con anterioridad, una expresion regular siempre equivale a la cadena mas larga posible que empieze tan cerca del principio de la linea como se pueda. por ejemplo, dado la cadena siguiente :

Este (tapete) no es lo que fue (hace mucho tiempo), no te parece?

La expresion **/E.*te/** equivale a:

Este (tapete) no es lo que fue (hace mucho tiempo), no te parece?

y **/(.*)/** equivale a:

(tapete) no es lo fue (hace mucho tiempo)

sin embargo **/([^)]*)/** equivale a:

(tapete)

UNA EXPRESION REGULAR NO EXCLUYE A OTRA

Si una expresion regular se compone de 2 expresiones regulares, la primera correspondera a una cadena lo mas larga posible, pero no excluira una equivalencia de la segunda. Dada la cadena siguiente:

contando cantatas, contando más y más

La expresión **/c.*ant/** equivale a :

contando cantatas, contando

y **/c.*ant cantat/** equivale a :

contando cantat

EXPRESIONESREGULARES VACIAS

Una expresion regular vacia siempre representa la ultima expresion regular usada. por ejemplo, si se desea sustituir la palabra marlon por la palabra enrique se hace de la siguinte manera :

s/marlon/enrique

y despues se desea hacer otra vez la misma sustitucuin puede emplearse el mandato

s//enrique/

La expreson regular vacia // representa la ultima expresión regular utilizada (/marlon/)

EXPRESIONES ENTRE CORCHETES

Los parentesis marcados, \(y \), pueden utilizarse para encerrar en corchetes unga expresion regular. La cadena que la expresion regular en corchestes hizo coincidir puede usarse a continuacion, como se explica más adelante en "Digitos Marcados". Una expresion regular no intenta equivaler a parentesis marcados. De esta forma, una expresion regular encerrada entre parentesis marcados corresponde a lo que la misma expresion regular, sin los parentesis, equivaldria

La expresion /\(rexp\)/ equivale a lo que /rexp/ equivaldria
y /\(b*\)c/ equivale a lo que /ab*c/ equivalaria

Los parentesis marcados pueden anidarse. La expresion siguiente consiste en 2 expresiones encerradas en corchetes, una dentro de la otra

/\([a-z]\(A-Z]*\)\)/

Las expresiones entre corchetes se reconocen solo por los \(de apertura, de modo que no hay ambiguedad en su identificacion

LA CADENA DE REPOSICION

Las expresiones regulares se utilizan como cadenas de busqueda dentro de mandatos de sustitucion en algunos editores (vi y sed). Para representar los cadenas coincidentes dentro de la cadena de remplazo correspondiente, pueden usarse 2 caracteres especiales, los signos & y los digitos marcados (\n)

SIGNOS &

Dentro de una cadena de remplazo, un signo & asume el valor de la cadena con la que la cadena de busqueda (la expresion regular) coincidio. Por ejemplo, el siguiente mandato de sustitucion rodea una cadena de uno o mas numeros con NN. El signo & en la cadena de reposicion equivale cualquier cadena de numeros que la expresion regular (cadena de busqueda) coincidio. Se requieren 2 definiciones de clase de caracteres porque la expresion regular [0-9]* a cero o mas ocurrencias de un digito, y cualquier cadena de caracteres es cero o mas ocurrencias de un digito.

```
s/[0-9] [0-9]*/NN&NN/
```

DIGITOS MARCADOS

Dentro de la propia expresion regular, un digito marcado toma el valor de la cadena que la expresion regular encerrada entre corchetes, que comienza con el n-ésimo \(, hizo coincidir

En una cadena de reposicion, un digito marcado (\n) representa la cadena que la expresion regular encerrada entre corchetes (porcion de la cadena de busqueda), que comienza com el n-ésimo \(, hizo coincidir

Por ejemplo, puede tomarse una lista de personas de la forma

apellido, primer nombre   y darle el formato siguiente

primer nombre, apellido

con el mandato siguiente (del editor vi) es posible

```
1,$s/\([^,]*\), \(.*\)/\2\1/
```

Este madato direcciona todas las lineas del archivo (1,$). El o los mandatos de sustitucion utilizan una cadena de busqueda y una cadena de remplazo delimitadas por las diagonales invertidas. La primera expresion regular entre corchetes dentro de la cadena de busqueda , \([^,]*\). corresponde a lo que equivaldria la misma expresion regular sin corchetes, [^,]* Esta expresion regular es igual a una cadena de cero o mas caracteres que no contengan una coma (el **apellido**). Despues de la primera expresion regular entre corchetes se encuentra una coma y un espacio que equivalen a si mismos. La segunda expresion entre corchetes \(.*\) corresponde a cualquier cadena de caracteres (el **primer nombre**)

La cadena de reposicion consiste en lo que la segunda expresion regular entre corchetes (\2) hizo coincidir seguido de un espacio y lo que la primera expresion regular entre corchetes hizo coincidir (\1)

CAPITULO 5

SINTAXIS DE ALGUNOS COMANDOS DEL SISTEMA OPERATIVO UNIX/XENIX

sed Empieza un editor de flujo

Sintáxis

sed [-e script] [-f sarchivo] [-n]

sed es un editor de flujo no interactivo, éste pasa los archivos que se especifican directo a un archivo que se haya creado y que contenga comandos de edición. Se debe crear primero el archivo conteniendo los comandos de edicion y luego ejecutar el comando **sed**.

Se puede especificar tambien sobre una linea de comandos que se editen los comandos no contenidos en el archivo de edición.

Las opciones de **sed** son:

-n Suprime la salida de default

-e script Causa que cualquiera de los siguientes -e sean ejecutados en los archivos que se especifiquen en la linea de comandos de **sed** . Por ejemplo si se escribe **-e d/java** todos las referencias a java en el archivo especificado deben ser borrados. Si la entrada solamente es un comando de edicion sobre la linea del comando **sed**, omitase **-e**.

-f sarchivo...Causa que **sed** pase los archivos especificados directo a el archivo sarchivo. Se puede crear sarchivo para que contenga los comandos de edicion que se quieran, entonces el comando **sed** es corrido referenciandose a el archivo con la opción -f

Cuando se crea sarchivo, se inserta cada comando sobre una linea separada en la forma siguiente:

[dirección [,dirección]] función [argumento]

donde direccion es igual a :

- Una linea opcional de dirección o 2 lineas direccionadas

- Un signo de $ (dolar) que significa la última linea de direccióy de salida

- Un contexto direccionado (una cadena de datos encerrada en ' ' a ser localizada en los archivos que se van a editar)

Los siguiente es cierto si el contexto direccionado es de la forma siguiente :

- La construcción \?expresion regular? (donde ? es cualquier caracter) es identico a /expresión regular/

- La secuancia de escope \n hace juego con una nueva línea insertada en el patron espaciado.

- Un punto (.) hace juego con cualquier caracter excepto con la terminación nueva línea (\n) de el patrón espaciado.

- Una línea de comando sin direcciones selecciona cada patrón espaciado

- Una linea de comando con una direccion selecciona cada patrón espaciado que haga juego con una direccion.

- Una linea de comando con 2 direcciones selecciona el rango desde el primer patron espaciado que haga juego con la primera dirección hasta el siguiente patrón espaciado que haga juego con la segunda dirección (si la segunda direccion es un número menor que o igual a la primera linea númerada seleccionado, solamente una linea es seleccionado) despues de eso el proceso es repetido viendo otra vez por la primera direccion.

función = Una simple letra de operación en la edicion, como la letra **d** que borra o **s** para sustituir.

argumento = Un parámetro final opcional (la s (de sustitucion) es solamente la operación que puede ser en esta posicion en una línea de comando).

En la siguiente lista de funciones, el máximo número de dirècciones permisibles para cada funcion es indicada en parentesis.

(1)a\ texto
Añade texto, poniendo sobre la salida antes de leer la siguiente línea de entrada.

(2)b label
Se ramifican : el comando de referencia a label si label es vacio se dirige a el fin del escrito.

(2)c\ texto Cambia texto , borra el patrón espaciado y entonces añade texto. Con las direcciones 0 o 1, o en el fin de la segunda dirección, pone el texto sobre la salida y empieza el ciclo siguiente

(2)d
Borra el patron espaciado y empieza el ciclo siguiente.

(2)D
Borra el segmento inicial de un patrón espaciado hasta hasta la primera nueva línea y empieza el ciclo siguiente.

(2)g
Remplaza el contenido de el patron espaciado con el contenido de el patron retenido espaciado.

(2)G
Añade el contenido de el patrón retenido espaciado a el patrón espaciado.

(2)h
Remplaza el contenido de el patrón retenido espaciado con el contenido de el patrón espaciado.

(2)H
Añade el contenido de el patrón espaciado a el patron retenido espaciado.

(1)i\ texto
Inserta, poniendo texto sobre la salida estandar

(2)l
Lista el patron espaciado sobre la salida estandar con caracteres de no impresion turnados de en 2 digitos ascii y lineas largas doblados.

(2)n
Copia el patron espaciado a la salida estandar, remplaza el patron espaciado con la siguiente linea de entrada.

(2)N
Añade la linea siguiente de la entrada a el patron espaciado con una nueva linea fijada.

(2)p
Imprime (copia) el patron espaciado sobre la salida estandar

(2)P
Imprime (copia) el segmento inicial de el patron espaciado hasta la primera nueva linea a la salida estandar.

(1)q
Sale de sed direccionandose a el fin del script y no empieza un nuevo ciclo.

(2)r rfile
Lee el contenido de rfile y pone ellos sobre la salida antes de leer la siguiente entrada de linea.

(2)s/expresion regular/remplazo/flags
Substituye la cadena expresión regular por la cadena remplazo, cualquier caracter puede ser usado poniendo / (para mas detalles ver el comando de edición **ed**), flags puede ser cero o más de :

g Substitución global por todas las instancias de las expreciones regulares de preferencia que sean más de 1.

p Imprime el patrón espaciado, si un remplazo es ejecutado

(2)t label
Se direccionan por 2 puntos (:) el comando es llevado a label, si cualquier substitución tiene que ser echa desde entonces la más reciente lectura de una linea de entrada o ejecución de un comando t,

si label es vacio t se direcciona a el fin del script.

(2) w warchivo
escribe el patrón espaciado en el archivo warchivo.

(2)x
intercambia el contenido de el patrón y los espacios retenidos

(2)y/cadena1/cadena2/
Remplaza todas las ocurrencias de caracteres en cadena1 con los correspondientes caracteres en cadena2. El largo de cadena1 debe ser igual a el largo de cadena2.

(2)!funcion
Aplica la funcion (o grupo de setencias si función empieza con {) solamente las líneas no seleccionadas por la(s) dirección(es).

(0):label
Este comando es nada; éste lleva a label para los comandos b y t a direccionarse.

(1)=
Pone el número de línea actual sobre la salida estandar de una linea.

(2){
Ejecuta los siguientes comandos hasta la llave (}) Solamente cuando el patron espaciado es seleccionado.

(0)
Un comando vacio es ignorado.

El texto del argumento consiste de una o más líneas.

awK

Invoca un editor de procesamiento de patrones

Sintáxis.

awK [-Fc] [-f archivo] ['programa'] [archivo]

El comando **awK** explora cada entrada de archivo por líneas y éstas son comparados con los patrones especificados en el programa. Con cada patrón en un programa, estos pueden ser asociados con una acción que puede ser ejecutada, cuando una línea de un archivo hace juego con el patrón. Los patrones determinados pueden aparecer literalmente como 'programa' o en un archivo especificado como -f archivo. La cadena programa debe ser encerrada en comillas simples ('programa') para proteger este desde el shell. La opción -Fc causa que c sea una etiqueta de separación.

Los archivos son leídos en orden; si estos no son archivos, la entrada estandar es leida. El nombre de archivo '-' significa la entrada estandar.

Cada línea es comparada con la porción del patrón de cada sentencia acción-patron; la acción asociada es ejecutada por cada patron ya comparado.

Una línea de entrada es hecha arriva de los campos separadores de campos por espacios. los campos son denotados $1, $2, ..., ; $0 se refiere a la línea entera.

Una sentencia acción-patron tiene la forma:

patron {acción}

Una {acción} ausente significa imprimir la línea; una ausencia de patrones siempre hace juego con todas y cada una de las líneas.

Una accion es una secuencia de sentencias que pueden ser de la siguiente forma:

```
if ( condición ) setencia [ else sentencia ]
while ( condición ) sentencia
for ( expresión ; condisión ; expresión ) sentencia
  break              # causa la salida inmediata
  continue           # para principiar la siguiente interacción
  { [ sentencia ] ... }
    variable = expresión
    print [ lista-expresión ] [ > expresión ]
    printf format [ , lista-expresion ] [ > expresión ]
    next             # para que se lea la siguiente línea de entrada
    exit             # origina la trasferencia inmediata al patrón END
```

Las sentencias son terminadas por punto y coma (;), nuevas líneas o llaves derechas (})

Una lista de expresiones vacias es colocada por la línea entera.

Las expresiones toman valores de cadena o valores númericos apropiadamente y son usados varios operadores.

+, -, *, /, y %

y la concatenación (indicada por un blanco)

Otras expresiones también disponibles.

Operadores relacionales	>, >=, <, <=, ==, !=
Operadores de incremento	++, --
Operadores de asignación	=, +=, -=, /=, %=
Operadores boleanos	||, &&, !

Las variables pueden ser escalares, arreglos de elementos (denotados por var[i]) o campos, las variables son inicializadas con cadenas nulas y los arreglos pueden ser cualquier cadena, no necesariamente numericas, estos se asignan por una forma de memoria asociativa, las constantes de cadena son encerradas por dobles comillas

La impresion de sentencias son impresas en la salida estandar (o en un archivo si el signo > es presentado) separados por el actual campo separador de salida y terminado por el registro separador de salida. la sentencia printf formatea estas listas de expresiones acorde a el formato (ver printf)

La funcion incorporada **length** retorna el largo de sus argumentos como una cadena, o de la linea entera existieran argumentos. También existen las siguientes funciones incorporadas.

exp, log, sqrt e int

La ultima funcion trunca sus argumentos a un entero.

La funcion **substr(s,m,n)** regresa la subcadena de la cadena s que empeza en la posicion m y termina en la posición.

La funcion **sprintf(fmt , expr , expr , ...)** formatea las expresiones acordes al formato de **printf(3)** dados por fmt y retorna la cadena resusltante

Los patrones son combinaciones arbitrarias boleanas **(!, ||, && y parentésis)** de expresiones regulares y expresiones relacionales, las expresiones regulares pueden ser encerrados por el simbolo / y son como el comando **egrep.**

Las expresiones regulares separados en un patrón se piden en la linea de entrada.

Las expresiones regulares también pueden ocurrir en expresiones relacionales.

Un patron puede consistir de 2 patrones separados por una coma, en este caso, la acción es ejecutada por todas las líneas entre la primera ocurrencia de el primer patrón y la siguiente ocurrencia del segundo patrón.

Una expresión relacional es una de las siguientes:

expreción **matchop** expresión regular
expreción **relop** expresión

Donde un **relop** es cualquiera de los 6 operadores relacionales del lenguaje C (ya mostrados), **matchop** es cualquiera de los 2 simbolos o !

Una condición es una expresión aritmética, una expresión regular o una combinación de estos, los patrones especiales **BEGIN** y **END** pueden ser usados para capturar el control antes de que la primera linea linea de entrada sea leída y después de la última linea. **BEGIN** necesita ser el primer patrón y **END** el último.

Un simple caracter c puede ser usado para separar los campos por el empiezo del programa de la siguiente manera :

BEGIN { FS = c }

o por el uso de la opción -Fc, otros nombres de variables con especial significado incluye :

NF	número de campos en el registro de entrada
NR	número total de registros de entrada
FILENAME	nombre del archivo de entrada actual
OFS	cadena separadora de campo de salida (blanco por default)
ORS	cadena separadora de registro de salida (nueva línea por default)
RS	caracter separador de registro de entrada (nueva línea por default)
FS	caracter separador de campo (blanco y tabulador por default)

NOTAS: Las conversiones entre cadenas y números no es explicito, es decir, se forza a una expresión a ser tratada como un número añadiendo un cero y para forzar una expresión a ser tratada como una cadena, se concatena a ella "" . La entrada con espacio no es preservada a la salida si el campo es envuelta

EJEMPLOS

Imprime lineas largas de mas de 72 caracteres

```
awk 'length > 72 '
```

Imprime los 2 primeros campos en orden opuesto

```
awk '{ print $2, $1 }'
```

Suma la primera columna, imprime la suma y el promedio

```
awk '{ s += $1 }
END { print "La suma es", s, "El promedio es", s/NR }'
```

Imprime los campos en orden inverso

```
awk '{ for (j = NF ; j > 0 ; --j) print $j }'
```

Imprime todas las lineas cuyo primer campo sea diferente desde un previo.

```
awk ' $1 != prev { print ; prev = $1 }'
```

Imprime todas las lineas entre empiezo/término

```
awk '/empiezo/, /término/'
```

Imprime la hora y el minuto de la salida de el comando **date**

```
date | awk '{ print substr($4,1,5) }'
```

Inserta cada renglón de entrada en el arreglo llamado linea y luego es impreso en orden inverso.

```
awk '{ linea[NR] = $0 }
END  { for ( m=NR ; m > 0 ; m-- ) print linea[m] }' $*
```

bc Invoca una calculadora

Sintaxis

bc [-c] [-l] [archivo]

bc es un proceso interactivo por un lenguaje parecido al C, pero provee una alta presicion aritmética, este comando toma la entrada desde cualquier archivo dado, entonces lee la salida estandar. Sus opciones son :

-c Solamente compila y envia la entrada a la salida estandar

-l Espera por el nombre de una arbitraria presion de la libreria matemática.

La sintaxis para los programas de **bc** es de la forma siguiente :

L significa letras 'a - z'
E significa expresion
S significa sentencias

Los comentarios son encerrados en /* y */

Nombres :

Simples variables L
Arreglo de elementos L[E]
Las palabras 'ibase', 'obase' y 'scale'

Otros operadores :

(E) largo arbitrario de los números con el signo opcional y punto decimal.

sqrt(E)
length(E) número significativo de digitos decimales.
scale(E) número de digitos a la derecha del punto decimal.
l(E,...,)

Operadores de adición y sustracción	+, -
Operadores de multiplicacion	*, /, %, ^
Operadores prefijo y sufijo	++, --
Operadores relacionales	==, <=, >=, !=, <, >
Operadores de asignación	=, =+, =-, =*, =/, =%, =^

Sentencias

```
E
{ S,...;S }
if (E) S
while (E) S
for (E ; E ; E) S
sentencias nulas
break
quit
```

Definicion de funciones

```
define L ( L,...,L ) {
  auto L,...,L
  S; ... S
  return(E)
 }
```

Funciones de la libreria matematica con la opción -l

s(x)	seno
c(x)	coseno
e(x)	exponencial
l(x)	logaritmo
a(x)	arco tangente
j(n,x)	función de Besel

Todos los argumentos de una funcion son pasados por valor, el valor de una sentencia el cual es una expresión a ser impresa a menos que el operador principal sea una asignación, cualquier punto y coma (;) o nuevas lineas puede separar sentencias.

La asignacion de la escala (scale) influencia en el número de digitos a ser retenidos en las operaciones aritméticas de **bc**.

La asignación de ibase y obase determinan la entrada y la salida de la base del sistema númerico, respectivamente.

La misma letra puede ser usada simultaneamente en un arreglo, una función y una variable.

Todas las variables son globales en el programa y las variables definidas en 'auto' son interpretadas como variables locales en la función.

Cuando usamos arreglos como los argumentos de una funcion o definiendo estos automáticamente como variables, los corchetes vacios son seguidos por el nombre de un arreglo.

bc es un preprocesador actual del comando **dc**, el cual es invocado automaticamente a menos que la opcion -c (solamente compilado) sea presentado, si la opcion -c es presentada, la entrada de **dc** es enviada a la salida estandar

ejemplo

```
/* programa calculo  uso la calculadora bc */
/* forma de uso    bc calculo   */
/* Simulacion de la funcion exponencial e(x) de los 10 primeros digitos *

scale=20            /* definicion de 20 numeros decimales */
define e(x) {       /* definicion de la funcion con el nombre e(x) */
                    /* e(x) en este caso no es la funcion de libreria */
  auto a,b,c,i,s     /* definicion de variables locales */
                     /* los variables son de UN SOLO caracter */
  a=1                /* asignaciones numericas a las variables */
  b=1
  s=1

  for (j=1; 1==1; j++)
  {
    a=a*x
    b=b*j
    c=a/b
    if (c==0)
      return(s)
    s=+c
  } /* fin de j */
}   /* fin de e(x) */

/* principal */

for (m=1; m<=10; m++)
  e(m) /*llama a la funcion e(x),no es la funcion de libreria de bc */
quit
```

Para evitar el algoritmo anterior podemos usar la libreria matematica de la calculadora **bc** con la opcion **-l**

```
/* programa calculo2  uso la calculadora bc */
/* forma de uso    bc -l calculo2   */
/* llamo a la libreria matematica para imprimir los valores  del
/* exponencial e(x) de los 10 primeros digitos */

/* principal */

for (m=1; m<=10; m++)
  e(m) /* llamada a la funcion e(x) de la libreria matematica de bc */
quit
```

Para ambos ejemplos la salida para los 10 primeros valores es :

```
    2.71828182845904523526
    7.38905609893065022713
   20.08553692318766774083
   54.59815003314423907790
  148.41315910257660342091
  403.42879349273512260821
 1096.63315842645859926350
 2980.95798704172827474335
 8103.08392757538400770974
22026.46579480671651695759
```

Nota

La sentencia **for** debera de tener todas las 3 E
quit es interpretado cuando se lee, no cuando es ejecutado

Comando du   Sumariza el uso del disco duro

Sintaxis

du [-ars] archivo ! directorio

Descripción

Cuando se escribe du, el número total de bloques en el directorio actual o en cualquiera de los subdirectorios, son desplegados en la pantalla. Para obtener el número de bloques de un directorio específico, se escribe el nombre del archivo o el nombre del directorio.

Opciones

-a Despliega el número de bloques por cada archivo generado

-r Genera un mensage por cada directorio o archivo que no puede ser leido.

-s Despliega solamente el gran total

Notas

Un archivo que tenga 2 ligas es solamente contado una vez.
Si un archivo tiene muchos archivos distintos ligados, du cuenta el exceso de los archivos más de una vez.

CAPITULO 6

PROGRAMACION DE COMANDOS CON EL LENGUAJE SHELL

```
# Uso:  copyto Directorio Archivo ...

# Copia los Archivo ... a un  Directorio
# Directorio debe ser una ruta /usr/marlon
# 2 argumentos deben ser dados y el 1/o debe ser una ruta o
# un Directorio
# y los demas argumentos son archivos
#
if test $# -lt 2
     then echo "Uso $0 Directorio Archivo
     ..."
elif test !  -d $1
     then echo "$0:  $1 no es un directorio";
else dir=$1; shift
     for eachfile
     do cp $eachfile $dir
     done
fi       # fin de los if anidados
```

comentario: este procedimiento usa el comando **if** anidado en diferentes partes del programa. El ciclo **for** hace iteracciones de todos los argumentos pasados a el programa pero el primero (Directorio) es cambiado, por lo tanto Archivo es puesto como $1 (shift)

```
# Uso:  draft Archivo s)
# Imprime las paginas del manual de la impresora diablo

for i in $*
      do nroff -man $i | lpr
done
```

```
# Uso:  edfind archivo argumento

# busca la ultima ocurrecia en archivo de una linea
# cuando esta es encontrado y haga juego con argumento
# 3 lineas son impresas
#
ed - $1 << -EOF
     ?-$2?
     -,+p
     q
EOF
```

comentarios: Lo anterior ilustra el uso del editor de textos **ed** al cual se le introducen valores de variables desde el shell

```
# Uso : edlast archivo

# imprime la ultima linea de un archivo
# y luego lo borra
    ed - $1 <<\!
    $p          # imprime la linea
    $d          # borra la linea
    w           # grava el archivo
    q           # salir del archivo
!
echo linea borrada
```

comentarios: las variables de sustitucion esta prohibidas dentro de la entrada de un texto por esta razon se usa \!

```
# Uso: fsplit file1 file2
# lee la entrada estandar y divide esta en 3 partes
# 1 añadiendo cualquier linea menor a una letra a file1
# 2 añadiendo cual'quier linea que contenga digitos a file2
# 3 arroja el resto
count=0 gone=0
while read next
do
    count=`expr $count + 1`
    case "$next" in
    *[A-Za-z]*)
        echo "$next" >> $1 ;;
    *[0-9]*)
        echo "$next" >> $2 ;;
    *)
        gone=`expr $gone + 1`
    esac
done
echo "$count lineas leidas , $gone al file3"
```

comentarios: cada iteracion lee una linea desde la entrada y analiza esta, el ciclo termina solamente cuando **read** encuentra el fin de archivo.

```
# Uso : mkfiles pref [quantity]
# makes 'quantity' files, named  pref1, pref2, ...
# Default is 5 as determined on following line.
quantity=${2-5}
i=1
while test "$i" -le "$quantity"
do
     >$1$i
     i="`expr $i + 1`"
done
```

comentarios: el procedimiento anterior usa la salida redireccionada para crear archivos mayores de longitud cero. el comando **expr** es usado para contar las iteracciones del ciclo while

```
# Usage:  null files

# Crea cada file nombrado en archivos vacios
for i in $*
do
     >$i
done
```

comentarios: crea archivos vacios si estos no existen

```
# Uso: phone iniciales ..

# imprime el numero telefonico de la
# gente cuando son dados sus iniciales
echo 'iniciales area telefono'
grep "^$1" <<-END
	jfk   1234    999-2345
	[illegible]   2234    583-2345
	[illegible]   3342    988-1010
	[illegible]   4567    555-1234
END
```

comentarios: este procedimiento es un ejemplo del uso de una pequeña base de datos

```
# uso: textfile
	if test "$1" = "-s"
	then
	# Return condition code
	shift
	if test -z "`$0 $*`" # checa el valor retornado
	then
			exit 1
		else
			exit 0
		fi
	fi

	if test $# -lt 1
	then	echo "$0: Uso: $0 [ -s ] file ..." 1>&2
		exit 0
	fi
	file $* | fgrep ' text' | sed 's/:.*//'
```

para determinar cuales archivos en un directorio contienen solamente informacion textual, textfile filtra la lista de argumentos a otros comandos. Por ejemplo la siguiente linea de comando imprime todos los archivos de texto en el directorio actual

```
pr `textfile *` | lpr
```

este procedimiento tambien usa la bandera -s el cual nos dice si el archivo especificado es un archivo de texto

```
# uso: writemail mensaje usuario

# si el usuario esta en el sistema
# escribe un mensaje a su terminal
# otro caso le envia correo electronico
echo "$1" |{write "$2" || mail "$2" ;}
```

comentarios: aqui se muestra el uso de los comandos de agrupacion. el mensaje $1 es pasado a el comando **write** y si este fallo entonces es enviado al comando **mail**

```
# comando c4 para compilar programas en informix-4gl
# y no poner el .4gl
# uso        c4 archivo
clear
i=1
#while test "$i" -le 20
#do
   #echo " \n "
   #i=`expr $i + 1`
#done
echo "                        [7m [5m Compilando a ejecutable [0m $1.4gl"
#while test "$i" -le 20
#do
   #echo -n "."
   #i=`expr $i + 1`
#done
time c4gl -s -o $1.4ge $1.4gl    # orden de compilacion
echo ""
echo ""
echo "                                 Borrando a :"
echo ""
echo "                                 $1.o"
echo "                                 $1.c"
echo "                                 $1.ec"
rm $1.o
rm $1.c
rm $1.ec
for j in 1 2
do
  echo " "
done
echo "                  [5m                         [m   [0m "
echo " [7m_Dbs_Dbs  [0m"
```

```
/* File s80.c
 * Autor : desconocido                            22 de diciembre 1988
 *
 * Nota :
 *   Este programa se diseño para trabajar juntamente con el more interno
 *                        lo que hace es esperar a que el usuario presione
 *   alguna tecla en su terminal, para cambiarla a modo de 80 columnas.
 *   Supone que se esta trabajando en una terminal altos5 a 132 cols.
 */

    #include <stdio.h>
    #define ESC 27

    main()
    {
     FILE  *file_tty, *fopen();
     char  ans;

     if( (file_tty = fopen("/dev/tty","r")) == NULL ) {
          printf("%c[35h%c[?3l", ESC, ESC);
          exit(1);
     }
     printf(" Retn. para continuar ");
     ans = fgetc( file_tty );
     printf("%c[35h%c[?3l", ESC, ESC);
     fclose(file_tty);
     exit(0);
    }



# File more    Este programa cambia a modo de 132 columnas si la terminal es
#              una alt5, tambien permite ordenar los reportes mandados por
#              un pipe desde un programa 4gl. Para correr necesita del prog.
#                     s80.c compilado a s80 (cc s80.c -o s80).

# uso more       (no es el more del sistema)             22 de diciembre 1988

    if [ "$TERM" = "alt5" -o "$TERM" = "altos5" ]
    then
         TERM=alt5
         export TERM
         awk '
         BEGIN { ESC = 27
             CR  = 13
             printf("%c[35h%c[?3h", ESC, ESC)    # escribe a 132 columnas
               }
     { printf("%s %c\n", $0, CR ) }' $1 | /usr/bin/more  # agrega el CR a c/linea
     s80             # pide un retn. y cambia a 80 columnas, ver s80.c
    else
         /usr/bin/more $1
    fi
    exit 0
```

```
# load : Crea programas para subir los respaldos
# Uso : load base_de_datos archivo.prn tabla num_campos

basename=$1      # := nombre de la base de datos
filename=$2   # := nombre de un archivo ascii de respaldo sin el .prn
tabla=$3         # := tabla a la que pertenece la informacion.
num_campos=$4   # := columnas de dicha tabla
if [ -s $filename.prn ]
then
   echo "# Para cargar los datos ejecute este programa. sh this file
   echo $filename.ld1 1>&2
   cat >$filename.ld1 <<'End of $filename.ld1'
   dbload -d $basename -c $filename.ld2 -l $filename.eld
   End of $filename.ld1
   echo $filename.ld2 1>&2
   cat >$filename.ld2 <<'End of $filename.ld2'
   FILE $filename.prn DELIMITER '|' $num_campos;
   INSERT INTO $tabla;
   End of $filename.ld2
   echo $filename.prn 1>&2
   cat >$filename.prn <<"End of $filename.prn" >$filename.sld
   cat $filename.prn |
   sed -n '1,$ s/^/ |
   '$/|/g' >> $filename.sld
   echo "End of $filename.prn
   chmod +x $filename.ld1
   $filename.ld1
   rm $filename.ld1 $filename.ld2
   rm $filename.eld $filename.prn* 2> $filename.sld
   chmod +x $filename.sld
   echo "Voy A Ejecutar el Load y a Cargar a la Tabla "
   echo $tabla
   $filename.sld
else
    echo "No existe el archivo de respaldo. Error."
fi
```

```
#
# programa vo
# sirve para no poner el .4gl a el nombre de un programa y evaluarlo
# si el programa no fue modificado entonces no se efectuara ninguna compilacion
# elaborado el 27/11/89
# autor Cz

clear
if test $# = 0
  then
    echo;echo;echo
    echo '                              Necesito un argumento '
    echo
    echo '                               usar    vo filename'
else
  if test -f "$1.4gl"
  then
    cp $1.4gl $1.cp
  else
    programa="#  PROGRAMA .......... :  $1.4gl "
    fecha="# `date '+ FECHA ............   : %d/%m/%y`'"
    hora="# `date '+ HORA .............   : %H:%M:%S`'"
    usr="#  USUARIO ........... : `whoami`"
    dir="#  DIRECTORIO ....... : `pwd`"
    term="#  TERMINAL ......... : `tty`"
    autor="#  AUTOR ............ : "
    echo $programa > $1.4gl
    echo $fecha >> $1.4gl
    echo $hora >> $1.4gl
    echo $usr >> $1.4gl
    echo $dir >> $1.4gl
    echo $term >> $1.4gl
    echo $autor >> $1.4gl
    cp $1.4gl $1.cp
  fi
  vi +/"AUTOR" $1.4gl
  if test `wc -l < $1.4gl` -eq 7
  then
    rm $1.4gl
    echo "                    Programa $1.4gl    Borrado por falta de lineas"
    echo;echo
    exit 0
  fi
  cmp -s $1.4gl $1.cp
  if test "$?" = 1
  then
    exe $1
    if test "$?" = 1
    then
      echo "                   $1.4gl  no es un Progama en Informix 4gl "
      echo;echo
      exit 0
    fi
```

```
    while test -f "$1.err"
    do
      vi +/'|' $1.err
      sed '/^|/d' $1.err > $1.4gl
      exe $1
    done
    echo;echo;echo;echo;echo
    echo "                    voy a ejecutar el programa    $1.4gl "
    sleep 3
    $1.4ge
  fi
  rm $1.cp
fi
```

```
# PROGRAMA .......... : TAR.sh
# FECHA ............. : 30/05/90
# HORA .............. : 11:37:57
# USUARIO ........... : root
# DIRECTORIO ........ : /usr/bin
# TERMINAL .......... : /dev/tty134
# AUTOR ............. : Czeraak andrade marlon
# este programa hace todo lo referente a los respaldos ya sea por cintas o
# disketts con la mayoria de las opciones de tar y dos
# el tama~o de bloques para las sucursales es 126
# el tama~o de bloques para la matriz  es 1024


disketts() {
while true
do
  disk_pant='

                                        Disketts


                                1 - Copiar un diskett a otro

                                2 - Formatear

                                3 - Respaldar (del CPU al diskette)

                                4 - Restaurar (del diskette al CPU)

                                5 - Ver el contenido

                                0 - Fin del menu

'
clear
echo "$disk_pant"
echo -n "                             Seleccione Una opcion "
read answer
clear
case "$answer" in
        1) fcopy
           ;;
        2) format
           ;;
        3) resp_disk
           echo;echo
           echo -n "                             Return  para continuar "
           read answer
           ;;
        4) rest_disk
           echo;echo
           echo -n "                             Return  para continuar "
           read answer
           ;;
        5) echo;echo
```

```
          echo " Leyendo Todo el Diskette "
          echo;echo
          tar tv
          echo;echo
          echo "                          He terminado de leer"
          echo;echo
          echo -n "                         Return  para continuar "
          read answer
          ;;
        0) echo
          break
          ;;
      "") ;;   # default mode
       *) echo "Solo de 1  2   3  4  5  o   0."
          sleep 2
          continue
          ;;
esac
done
}

resp_disk() {
dir=`pwd`
SCREEN='




                    Respaldar Diskette



           1 - Todos los archivos del directorio :
               $dir

           2 - Un(os) archivo(s) de una ruta especifica

           0 - Fin del menu

'
clear
echo "$SCREEN"
echo -n "                  Seleccione una opcion : "
read answer
clear
SCREEN='




                 Guardando la Informacion
```

```
                1 - Con reinicializacion del diskette

                2 - Final del ultimo archivo

                0 - Fin del menu

case "$answer" in
        1) echo "$SCREEN"
           echo -n "                          Selecione una opcion : "
           read ans
           clear
           if test $ans -eq 1
           then
             echo;echo
             echo " Respaldando en diskette con reinicializacion"
             echo " el directorio $dir"
             echo;echo
             tar cv $dir
           elif test $ans -eq 2
             then
             echo;echo
             echo " Respaldando en diskette al final del ultimo archivo "
             echo " el directorio $dir"
             echo;echo
               tar rv $dir
           fi
           ;;
        2) echo "$SCREEN"
           echo -n "                          Selecione una opcion : "
           read ans
           echo;echo
           echo "deme los nombres de los archivos separados por espacios"
           echo
           read archivos
           clear
           if test $ans -eq 1
           then
             echo;echo
             echo " Respaldando en diskette con reinicializacion"
             echo " la ruta $archivos"
             echo;echo
             tar cv $archivos
           elif test $ans -eq 2
             then
             echo;echo
             echo " Respaldando en diskette al final del ultimo archivo "
             echo;echo
             echo " la ruta $archivos"
               tar rv $archivos
           fi
           ;;
        0) echo
           break
           ;;
        "") ;;   # default mode
```

```
        *) echo "Solo de 1  2  o  0."
           sleep 2
           continue
           ;;
esac
echo;echo
echo "                              He terminado de respaldar"
}

rest_disk() {
dir=`pwd`
# cuando se respalda mas de 1 vez un mismo archivo estos estaran en el diskette
# es decir estara tantas veces como respaldos se hayan echo por lo que cuando de
# restaura un archivo bajaran todos los que esten en el orden en que fueron res-
# paldados y al final solo quedara el mas reciente

SCREEN="




                              Restaurar Disketts
                              ( de la Cinta al CPU)



                         1 - Todos los archivos del directorio
                                la ruta $dir

                         2 - Un(os) archivo(s) de uno(s) ruta(s) especifica

                         0 - Fin del menu

"
clear
echo "$SCREEN"
echo -n "                              Selecione una opcion : "
read answer
echo
case "$answer" in
        1) echo;echo
           echo" Restaurando toda la Ruta $dir"
           echo
           tar xv $dir
           ;;
        2) echo "deme los nombres de los archivos separados por espacios"
           echo
           read archivos
           clear
           echo;echo
           echo" Restaurando (del Diskette al CPU) la Ruta "
           echo" $archivos"
           echo
           tar xv $archivos
           ;;
```

```
        0) echo
           break
           ;;
       "") ;;   # default mode
        *) echo "Solo de 1  2  o  0."
           sleep 2
           continue
           ;;
esac
echo;echo
echo "                          He terminado de restaurar"
}


cintas() {
while true
do
  cint_pant='

                                        Cintas



                    1 - Respaldar (del CPU a la cinta)

                    2 - Restaurar (de la cinta al CPU)

                    3 - Ver el contenido

                    4 - Reemplazado/Borrado

                    0 - Fin del menu

'
clear
echo "$cint_pant"
echo -n "                         Seleccione Una opcion "
read answer
clear
case "$answer" in
        1) resp_cint
           echo;echo
           echo -n "                              Return  para continuar "
           read answer
           ;;
        2) rest_cint
           echo;echo
           echo -n "                              Return  para continuar "
           read answer
           ;;
        3) echo;echo
           echo " Leyendo Toda la Cinta "
           echo;echo
           tar tvbf 126 /dev/rct
           echo;echo
           echo "                              He terminado de leer "
```

```
          echo;echo
          echo -n "                    Return para continuar "
          read answer
          ;;
       4) tapeutil
          echo;echo
          echo -n "                    Return para continuar "
          read answer
          ;;
       0) echo
          break
          ;;
      "") ;;   # default mode
       *) echo "Solo de 1  2   3  4  o  0  "
          sleep 2
          continue
          ;;
esac
done
}


resp_cint() {
dir=`pwd`
SCREEN="



                    Respaldar Datos



               1 - Todos los archivos del directorio
                   $dir

               2 - Un(os) archivo(s) de una ruta especifica

               0 - Fin del menu

    "
clear
echo "$SCREEN"
echo -n "                    Seleccione una opcion : "
read answer
case "$answer" in
       1)   clear;echo;echo
            echo " Respaldando en cinta con sincronizacion"
            echo " el directorio $dir"
            echo;echo
            tar cvf 126 /dev/rct  $dir
          ;;
       2) echo "deme los nombres de los archivos separados por espacios"
```

```
          echo
          read archivos
          clear
          echo;echo
          echo ' Respaldando en cinta con reinicializacion'
          echo ' la ruta $archivos'
          echo;echo
          tar cvbf 126 /dev/rct  $archivos
          ;;
        0) echo
          break
          ;;
       "") ;;   # default mode
        *) echo 'Solo de 1  2  o  0.'
          sleep 2
          continue
          ;;
esac
echo;echo
echo '                          He terminado de respaldar'
}


rest_cint() {
dir=`pwd`
SCREEN='



                      Restaurar Datos (de la Cinta al CPU)



                        1 - Todos los archivos de la Cinta
                            la ruta $dir

                        2 - Un(os) archivo(s) de una(s) ruta(s) especifica(s)

                        0 - Fin del menu

    '
clear
echo "$SCREEN"
echo -n '                        Selecione una opcion : '
read answer
clear
case "$answer" in
        1) echo;echo
           echo ' Restaurando TODO el contenido de la Cinta en el CPU'
           echo ' la ruta $dir'
           echo;echo
           tar xvbf 126 /dev/rct $dir
           ;;
```

```
        2) echo 'dame los nombres de los archivos separados por espacios'
           echo
           read archivos
           clear
           echo;echo
           echo ' Restaurando el contenido de la Cinta en el CPU'
           echo ' la ruta $archivos'
           echo;echo
           tar xvbf 126 /dev/rct $archivos
           ;;
        0) echo
           break
           ;;
       "") ;;   # default mode
        *) echo 'Solo de 1  2  o  0.'
           sleep 2
           continue
           ;;
esac
echo;echo
echo '                              He terminado de restaurar'
}

# programa principal
while true
do
clear
programa='   PROGRAMA .......... :  $0 '
   fecha='  `date '+ FECHA .............    : %d/%m/%y '`'
    hora='  `date '+ HORA ..............     : %H:%M:%S '`'
     usr='  USUARIO ........... : `whoami`'
     dir='  DIRECTORIO ........ : `pwd`'
    term='  TERMINAL .......... : `tty`'
   autor='   AUTOR ............. : Marlon Czermak'
    echo $programa
    echo $fecha
    echo $hora
    echo $usr
    echo $dir
    echo $term
    echo $autor

SCREEN='


              UTILERIAS PARA CINTAS y/o DISKETTS    version 1.0



                          1 - Disketts
```

2 - Cintas

0 - Quit

```
# trap 'echo tar.sh aborted;exit 1' 1 2 3 15
echo "$SCREEN"
echo -n "                    Selecione una opcion : "
read answer
clear
  case "$answer" in
         1) disketts
            ;;
         2) cintas
            ;;
         0) echo
            break
            ;;
        "") ;; # default mode
         *) echo "Solo de 1  2  o  0."
            sleep 2
            continue
            ;;
  esac
done
```

```
# programa DF.sh para conocer el espacio en disco
# autor Czernak
# creado 03/jul/90
#
echo
df -t $* |nawk '
{
  if ($1== "total:")
   {
     esp_tot=(($2*512)/1048576)-0.005       # espacio total libre en FS
     cont_esp_tot +=esp_tot
     printf "%-11s:\t%6.2f MB  de  %6.2f MB        (%5.2f%%).\n",
            directorio,espacio,esp_tot,(espacio/esp_tot)*100
   }
  else
   {
     directorio=$1
     bloques=$(NF-3)
     espacio=((bloques*512)/1048576)-.005
     cont_esp +=espacio
   } # fin de if
}    # fin de la rutina
END {
  printf "\n%-11s:\t%6.2f MB  de  %6.2f MB        (%5.2f%%).\n",
         "Total", cont_esp, cont_esp_tot, ((cont_esp/esp_tot)*100)
}'
```

```
# programa trad
# sirve para traducir ingles espanol de la base de datos Ingles
# trad es un prog que llama a el prog /usr/Ingles
# si una palabra no es encontrada el prog pregunta el significado de esta
# y lo inserta en el archivo /usr/rafa/Ingles
# autor marlon czermak

clear
echo;echo;echo
if test $# = 0
  then
    echo '                    Necesito un argumento '
    echo
    echo '                     usar   trad arg'
else
  echo ;echo
  echo -n '-------------------------------------------------------------'
  echo -n 'Diccionario Ingles - Espanol  con `wc -l < /usr/rafa/Ingles`  Palabras  Registradas'
  echo
  echo -n '-------------------------------------------------------------'
  echo;echo;echo
  grep -y " $1 " /usr/rafa/Ingles
  if test "$?" = 1
    then
      echo -n '                 $1   no registrado en el diccionario'
      echo;echo
      echo '                   dame el significado de $1 '
      read esp
      if test "$esp" = ""
        then
          echo '           $1    no fue insertado por falta de traduccion'
          echo
        else
          linea=" $1              - $esp "
          echo "$linea" >> /usr/czermak/Ingles
        fi
  fi
  echo;echo
  echo -n '--------------- Czermak -----------------------'
 echo
 fi
 echo;echo
```

```
# archivo Ingles
# contiene las palabras en ingles y su traduccion al espa"nol
#

 Accomplish       - Lograr, conseguir, culminar.
 Accuracy         - Presicion.
 Achieve          - llevar a cabo, conseguir,realizar.
 Ado              - Actividad, bullo,disturbio.
 Aesthetic        - Estetica.
 Allow            - Permitir, dejar , conceder.
 Along            - A lo largo.
 Although         - Aunque, bien que.
 Among            - Entre, en medio.
 Amount           - Cantidad, Suma, Importe.
 Appeal           - Acudir . Pedir ayuda, Implorar.
 Append           - A"adir,Agregar,anexar.
 Arrange          - Arreglar, acomodar.
 Array            - Arreglo, formacion.
 Available        - Disponible.
 Best               - superior, optimo, mayor
```

```
:
# PROGRAMA .......... : DU.sh
# FECHA ............. : 08/11/90
# HORA .............. : 16:00:22
# USUARIO ........... : root
# DIRECTORIO ........ : /usr/bin
# TERMINAL .......... : /dev/tty65
# AUTOR ............. : Carmen andrade morlan
#
# programa DU.sh para conocer el espacio en disco de cada directorio
# 1 Bloque = 512 Bytes     1024000 Bytes = 1 Mega Byte = 2048 = Bloques
# DU -a para lista el tamaño de cada archivo
# DU -s para lista el total de el o los directorios especificados
# DU -u para mostrar el tamaño de cada directorio de cada usuario

echo
if test "$1" = "-u"
then
  cd /usr
  directorios=`ls -1`
  for i in $directorios
  do
    if test -d "$i"
    then
      case "$i" in
        "DEMO" | "adm"     | "adm.PLUS" | "bin" | "admon" | "tmp" | "lib") : ;;
        "UII"  | "build"   | "include"  | "include.xenix" | "informix"   ) : ;;
        "ucs"  | "knowhow" | "raglmix"  | "admin" | "spool" | "sys"      ) : ;;
        * )
        du -s $i | awk '{
        file=$2
        esp=(($1*512)/1024000)+0.005 # esp ocupado MB
        if (esp < 1 && esp > 0)
          {
          esp=($1*512)                # esp ocupado Bytes
          printf "                     \t %8d         Bytes  de  %s\n", esp,file
          }
        else
          if ( esp >= 1.5 )
            printf "Someter a Revision\t\t   %3.2f  Mega Bytes  de  %s\n",esp,file
          else
            printf "                   \t\t   %3.2f  Mega Bytes  de  %s\n",esp,file
        }'   # fin de la rutina
      esac
    fi
  done
else
  du $* | awk '
   {
    arch=$2
    espacio=(($1*512)/1024000)+0.005 # espacio ocupado de cada archivo en MB
```

```
    if (espacio < 1 && espacio > 0)
      {
        espacio=($1*512)              # espacio ocupado de cada archivo en Bytes
        printf "%8d  Bytes  de  %s\n", espacio,arch
      }
     else
       printf "%3.2f Mega Bytes  de  %s\n", espacio,arch
   }'   # fin de la rutina
fi
echo
```

```
:
#
# usar LPR numero_de impresoare archivo(s)
# programa LPR para generar impreciones con fecha ,hora y nombre del programa
# autor marlon Cz
# elab: 20/6/89
#

trap 'echo;echo comando LPR abortado ;exit 1' 1 2 3 15

if test $# -eq 0
then
  echo ' Usar : $0  numero de impresora archivo(s)  '
  echo '        $0  -h    para ayuda... '
  exit
elif test '$1' = '-h'
then
  clear
  for j in 1 2 3 4 5 6 7
  do
    echo ''
  done
  echo '              este comando imprime los archivos enviados como parametros
        pero en la cabecera de cada imprecion estaran escritos los
        siguientes datos :

                     FECHA
                        dia/mes/año
                        HORA
                        hr:min:seg
                        nombre del usuario
                        terminal de trabajo
                        nombre del archivo'
  echo ''
  exit
elif test $# -ge 1
then
  echo
  until false
  do
    echo -n ' Los archivo(s) $* saldran por la impresora  lpr'
    read impresora
    case '$impresora' in
      [1-99] ) break ;;
           *) ;;
    esac
  done
  TTY=`tty`
  USR=`logname`
  DIR=`pwd`
  for i in $*
  do
    if [ -f '$i' ]
    then
      echo ' $i   procesandose en   lpr${impresora} '
      banner `date '+FECHA  %d/%m/%y%nHORA.%H.%M.%S'` ${USR} ${TTY} ${DIR} $i |
      lpr${impresora}
      cat $i |
```

```
        awk 'BEGIN { for (i=1; i<=4; i++)
                       print " "
                   }
           { if (NR % 59 == 0)
               {
                 print NR," ",$0
                 for (i=1; i<=9; i++)
                   print " "
               }
              else
                print NR," ",$0
           }' | lpr${impresora}
    else echo " El archivo $i no existe "
    fi
  done
fi
```

```
# PROGRAMA .......... : off.sh
# FECHA ............. : 19/11/90
# HORA .............. : 10:14:08
# USUARIO ........... : root
# DIRECTORIO ........ : /usr/bin
# TERMINAL .......... : /dev/tty65
# AUTOR ............. : czermak andrade marlon

# este programa se auto-apaga en un tiempo n  desde cualquier terminal o
# la consola maestra cuando solo exista un usuario, si en el tiempo especificado
# no se sale nadie se dara un killall a todos los procesos

trap 'echo;echo comando  off  abortado ;exit 1' 1 2 3 15

if [ `logname` != "root" ]
then
  echo;echo " Permiso Denegado ";echo
  exit 1
fi
clear
echo;echo
programa="   PROGRAMA .......... :  $0 "
   fecha="  `date '+ FECHA .............    : %d/%m/%y '`"
    hora="  `date '+ HORA ..............     : %H:%M:%S '`"
     usr="   USUARIO ...........  : `whoami`"
     dir="   DIRECTORIO ........ : `pwd`"
    term="   TERMINAL .......... : `tty`"
   autor="    AUTOR ............. : Ing marlon czermak"
    echo $programa
    echo $fecha
    echo $hora
    echo $usr
    echo $dir
    echo $term
    echo $autor

minuto_empieza=`date '+ %M'`
segundo_empieza=`date '+ %S'`
usr_inicio=`who | wc -l`
usr_fin=`who | wc -l`
echo;echo;echo
tiempo=-1
while test $tiempo -lt 0 || test $tiempo -gt 60 || [ "$tiempo" = "[a-zA-Z]*" ]
do
 echo -n "            Tiempo de apagado del sistema         [default 1 min] "
 read tiempo
 if test "$tiempo" = ""
 then
   tiempo=1
 fi
done
minuto_fin=`expr $minuto_empieza + $tiempo`
minuto_wall=`expr $minuto_empieza + 1`
```

```
if test $minuto_fin -gt 60
then
   minuto_fin=`expr $minuto_empieza - 60`
   hora_fin=``expr $hora_empieza + 1``
fi
if test $minuto_fin -le 9
then
  minuto_fin=0${minuto_fin}
fi
if [ $minuto_wall -le 9 ]
then
  minuto_wall=0${minuto_wall}
fi
WALL="

############################################################################


      Favor de salirse del sistema ya  que el equipo  se apagara en $tiempo min
      por presentar fallas de tipo Hadware

      Salve su informacion debidamente para evitar la perdida de esta ya que
      el servicio se reanudara hasta NUEVO AVISO


              De     CTRL r          Para que reaparezca su pantalla


                              Atentamente

                              Informatica


############################################################################

"
echo "$WALL" | wall
clear
echo;echo;date;echo
who
echo
echo " $usr_inicio usuarios trabajando "
minuto_cambio=`date '+ %M'`
while test $usr_fin -gt 1 && test $minuto_cambio -ne $minuto_fin
do
  usr_cambio=`who | wc -l`
  $minuto_cambio=`date '+ %M'`
  while (test $usr_fin -eq $usr_cambio) && (test $minuto_cambio -ne $minuto_fin)
  do
    usr_cambio=`who | wc -l`
    minuto_cambio=`date '+ %M'`
```

```
    if test $minuto_cambio -eq $minuto_wall
    then
        echo "$WALL" | wall
        clear
        echo;echo;date;echo
        who
        echo

        if test $usr_inicio -eq $usr_fin
        then
           echo " $usr_inicio usuarios trabajando "
        elif test $usr_inicio -gt $usr_fin
        then
           echo "  De $usr_inicio usuarios salio `expr $usr_inicio - $usr_fin` "
        elif test $usr_inicio -lt $usr_fin
        then
           echo "  De $usr_inicio usuarios entro `expr $usr_fin - $usr_inicio` "
        fi

        echo "  Quedan por salirse  $usr_fin"
        minuto_wall=`expr $minuto_wall + 1`

        if test $minuto_wall -le 9
        then
           minuto_wall=0${minuto_wall}
        fi
    fi
#   echo $minuto_empieza `date +%H%M%S` ''  $minuto_cambio $minuto_fin
  done
  usr_fin=$usr_cambio
  clear
  echo;echo;date;echo
  who
  echo
  if test $usr_inicio -eq $usr_fin
  then
    echo " $usr_inicio usuarios trabajando "
  elif test $usr_inicio -gt $usr_fin
  then
    echo "  De $usr_inicio usuarios salio `expr $usr_inicio - $usr_fin` "
  elif test $usr_inicio -lt $usr_fin
  then
    echo "  De $usr_inicio usuarios entro `expr $usr_fin - $usr_inicio` "
  fi
  echo "  Quedan por salirse  $usr_fin"
done

if test `tty` = "/dev/console"
then
   :
   shutdown 0
else
   :
   sync;sync;haltsys
fi
```

G L O S A R I O

G L O S A R I O

Acceso En lenguaje de computación, esta palabra se emplea con frecuencia para indicar leer archivo o escribir en él.

Actual (proceso, línea, carácter, directorio, evento, etc.) Elemento disponible de inmediato, trabajando o en uso. El proceso actual es aquel que controla el programa que se está ejecutando; la línea actual o el carácter actual es el sitio que señala el cursor; el directorio actual es de trabajo.

Administrador del sistema La persona responsable del mantenimiento del sistema.

Agregar al final Anexar algo al final de otra cosa. Anexar texto a un archivo significa añadir texto al final del archivo.

Ambiente de programación Lista de variables (y sus valores) disponible para el programa llamado. Véase (Ambiente y exportación de variables), en el capítulo 8, y (Sustitución de variables), en el capítulo 9.

Archivo Colección de información relacionada entre sí, que se identifica con un nombre de archivo. El sistema operativo UNIX considera los dispositivos periféricos como archivos, permitiendo, que un programa lea o escriba en un dispositivo igual que haría en un archivo normal.

Archivo especial Archivo que representa a un dispositivo. Existen tres tipos de archivos en el sistema UNIX: ordinarios, directorios y especiales (dispositivos).

Archivo invible Archivo cuyo nombre empieza con un punto. Reciben este nombre porque el mandato **ls** no suele listarlos. La opción **-a** de este comando despliega todos los archivos, incluyendo los invisibles.

Archivo ordinario Archivo que sirve para almacenar un programa, texto o datos, a diferencia de los archivos directorio y especial.

Argumento Número, letra o palabra que proporciona información a un programa cuando éste se ejecuta. El argumento de una línea de mandato es lo escrito a continuación del mandato mismo.

Arreglo Disposición de elementos (números o cadenas de caracteres) en una o más dimensiones. El **C Shell** puede almacenar y procesar arreglos.

ASCII Acrónimo de American Standard Code for Information Interchange. Es un código que usa siete bits para representar caracteres gráficos (letras, números y signos de puntuación) y de control. El **ASCII** puede representar tanto texto como progrmas fuente. Dado que es estándar, se usa con frecuencia para intercambiar información entre computadoras.

Existen extensiones del conjunto de caracteres **ASCII** que ocupan ocho bits, pero el de siete bits es el más común.

Barrido Movimiento de lineas en una terminal hacia arriba o hacia abajo, una linea cada vez.

Bit El bloque de información más pequeño que puede manejar un computador. Un bit es uno (on: activado) o un cero (off: desactivado).

Blanco Un espacio o un tabulador.

Buffer (Area de almacenamiento temporal) Area de memoria que guarda datos hasta que éstos puedan utilizarse. Cuando se escribe en un archivo de disco, el UNIX almacena la información en un buffer de disco hasta tener la suficiente para escribir al disco o hasta que éste se halle listo para recibir la información.

Buffer de trabajo Localidad de memoria donde los editores **ed** y **vi** almacenan texto mientras se está editando.

Byte Ocho bits de información. Un bite puede almacenar un carácter.

Cadena Una secuencia de caracteres.

Cadena nula Una cadena que podría contener caracteres, pero no los tiene.

Cambio (swap) Pasar un proceso de la memoria a un disco, o viceversa. El cambio de un proceso a un disco permite que otro proceso comience o continúe su ejecución.

Carácter alfanumérico Algún carácter comprendido entre la A y la Z (mayúscula o minúscula) o entre el cero y el nueve.

Carácter de | **CONTROL** | Un carácter que no es gráfico, es decir, que no es una letra, un número o un signo de puntuación. Se les llama caracteres de | **CONTROL** | porque suelen controlar un dispositivo periférico.

Return y **LINE FEED** son caracteres de **control** para un terminal o impresora.

La palabra **CONTROL** se escribe en mayúsculas en este libro porque es una tecla que aparece en la mayoría de los teclados de terminal. Puede aparecer como **CNTRL** o **CTRL**. Los caracteres de **CONTROL**, a menudo llamados caracteres no imprimibles, se representan con códigos **ASCII** menores que 32 (decimal).

Carácter especial Un carácter que no se representa a sí mismo a menos que esté marcado. También son caracteres especiales del **Shell** el asterisco (*) y el signo de interrogación (?).

Carácter imprimible Uno de los caracteres gráficos: letra, número o signo de puntuación.

Carácter no imprimible Véase carácter de control.

Carácter regular Un carácter que siempre se representa a sí mismo; no tiene significado especial.

Clase de caracteres Un grupo de caracteres que señala cuáles pueden ocupar una posición individual. Una definición de una clase de caracteres, por lo general, se encierra entre corchetes. La clase de caracteres determinada por [abcr] representa una posición de carácter que puede ser ocupada por a, b, c o r.

Código condicional Véase código de retorno.

Código de retorno Código que indica el estado que devuelve un proceso: éxito (suele ser un cero) o fracaso (un uno).

Concatenar Unir secuencialmente, o extremo con extremo. La utilidad **cat** del UNIX concatena archivos (los despliega uno después de otro).

Conducto (o pipe) Conexión entre dos programas, demodo que la salida estándar de uno se conecta con la entrada estándar del otro.

Control de dispositivo Progra que controla un dispositivo, como un terminal, una unidad de disco, o una impresora.

Corchetes Corchetes ([]) o signo de mayor que (>) y menor que (<).

Cursor Pequeño rectángulo o línea luminosa que sale en la pantalla del terminal e indica donde aparecerá el próximo carácter.

Darse de alta Obtener acceso a un sistema UNIX, respondiendo en forma correcta a las indicaciones **login:** y **passaword:**.

Darse de baja Dejar de usar ell terminal ein un sistema UNIX para q ue otro usuario pueda darse de alta.

Depurar Corregir un programa.

Diferencia de mayúsculas o minúsculas Capacidad de distinguir entre caracteres mayúsculos y minúsculos. A menos que se fije el parámetro **ignorecase**, el editor **vi** establece la diferencia.

Directorio Archivo directorio: contiene una lista de otros archivos.

Directorio de trabajo Directorio con el que se está asociado en un momento determinado. Los nombres de trayectoria relativos utilizados se basan en este directorio.

Directorio domicilio Directorio de trabajo utilizado al darse de alta en el sistema. El nombre de trayectoria de este directorio está almacenado en la variable **PATH** (**Bourne Shell) o path** (C Shell).

Directorio raíz El ancestro de todos los directorios y el comienzo de todos los nombres de trayectoria absolutos.

Dispositivo Véase dispositivo periférico.

Dispositivo de bloque Unidad de disco o de cinta. Un dispositivo bloque almacena información en bloques de caracteres. A diferencia de un dispositivo carácter, un dispositivo de bloque está representado por un archivo especial llamado archivo especial de bloque (block special file).

Dispositivo de carácter Un terminal, impresora o modem. Un dispositivo de bloque. El dispositivo de carácter está representado por un archivo especial llamado archivo especial de carácter (character special file).

Dispositivo físico Dispositivo, como una unidad de disco, separado física y lógicamente de otros dispositivos similares.

Dispositivo periférico Unidad de disco, impresora, terminal, graficadora u otra unidad de entrada/salida que puede conectarse al computador.

Elemento constituyente Parte básica, de un grupo un elementos de un arreglo numérico es uno de los números almacenados en el arreglo.

Elemento del nombre de trayectoria Uno de los nombres de archivo que componen el nombre de trayectoria.

Encabezado Parte de un formato que va en el extremo superior de una página.

Entrada Información que se alimenta a un programa desde un terminal u otro archivo.

Entrada estándar Archivo de donde un programa puede recibir datos. A menos que se indique al Shell lo contrario, la entrada estándar procederá del terminal.
Véase el capítulo 4.

EOF (FIN DE ARCHIVO) Acrónimo de End Of File; representa el fin de un archivo.

Espacio Carácter que aparece como la ausencia de un carácter visible. A pesar de que no se ve, es un carácter imprimible. Está representado por el código ASCII 32 (decimal).

Evento asíncrono Evento que no ocurre con regularidad o en sincronía con otro evento. En el UNIX, las señales son asíncronas; pueden ocurrir en cualquier momento porque pueden iniciarse por un número cualquiera de eventos irregulares.

Expresión Véase expresión lógica o expresión aritmética.

Expresión aritmética Grupo de números, operadores y paréntesis evaluables. Cuando se evalúa una expresión aritmética, se obtiene como resultado un número.

Expresión lógica Serie de cadenas de caracteres separados por operadores lógicos (<, >, !=, =, <=, >=). Después de evaluar una expresión lógica, el resultado puede tomar únicamente dos valores: falso o verdadero.

Expresión regular Cadena compuesta de letras, números y símbolos especiales que define a una o más cadenas. Véase el apéndice A.

Extensión del nombre de archivo Parte de un nombre de archivo que va después de un punto.

Hoja En una estructura de árbol, la parte final de una rama que no puede soportar otras ramas. Lo contrario de un nodo.

Identificador de proceso (PID) Acrónimo de Process Identification (identificación de proceso) que usualmente va precedido de la palabra **número**. El UNIX asigna un número PID único a cada proceso en el momento de iniciarlo.

Identificación de usuario (UID) Número asociado al nombre de un usuario. El archivo /**etc/ passwd** contiene una lista de los usuarios y sus UID asociados.
i-nodo Parte de la estructura del directorio que contiene información de un archivo.

Instalación Un computador en un lugar específico. Algunos aspectos del UNIX son dependientes de la instalación.

Justificar Expandir una línea hasta el margen derecho. Una línea se justifica aumentando el espacio existente entre palabras y a veces entre letras de la línea.

Línea de estado Última línea de la pantalla (por lo común la número 24).

Línea de mandato Línea de instrucciones y argumentos que ejecuta un mandato. Este término, por lo general, se refiere a una línea que se introduce en respuesta a una indicación del Shell.

Llaves de puntuación Existe la llave de apertura ({) y la de cierre (}).

Macro Instrucción sencilla que un programa sustituye por varias instrucciones (que suelen ser más complicadas). Un alias de C Shell es un macro.

Marcar Eliminar el significado especial de un carácter. Puede marcarse un carácter precediéndolo con una barra diagonal invertida o encerrándolo entre apóstrofos. Por ejemplo, el Shell expande el asterisco (*) en una lista de los archivos del directorio de trabajo. El mandato **echo** * o **echo '*'** despliega *.

Mayor que y menor que Hay un signo de mayor que (>) y otro de menor que (<).

Medio Véase ambiente de programación.

Mezclar Combinar dos listas ordenadas de modo que la lista resultante quede también ordenada.

Nodo En una estructura de árbol, el extremo de una rama que puede soportar otras ramas. Lo contrario es hoja.

Nombre de archivo Denominación de un archivo. Se utiliza para hacer referencia un archivo.

Nombre de trayectoria Relación de directorios separados por barras diagonales (/). Un nombre de trayectoria sirve para seguir una trayectoria a través de la estructura de archivos y así localizar o dentificar un archivo.

Nombre de trayectoria absoluto Trayectoria que comienza con el directorio raíz (/). Un nombre de trayectoria absoluto localiza un archivo sin tener en cuenta el directorio de trabajo.

Nombre de trayectoria relativo Un nombre de trayectoria que parte del directorio actual y no del raíz, como el nombre de trayectoria absoluto.

Nombre de trayectoria, último elemento de un La parte de un nombre de trayectoria que va despues de la última barra diagonal (/) o el nombre de archivo completo (si no hay diagonales). Nombre sencillo de archivo.

Nombre de usuario El nombre que se digita en respuesta a la indicacion **login:**. Con este nombre, los usuarios pueden enviar por correo (mail) o escribir (write) a otro usuario.

Nombre sencillo de archivo Un nombre de archivo que no contiene diagonales. Un nombre de archivo es la forma más simple de un nombre de trayectoria (el último elemento).

Núcleo (Kernel) El centro del sistem UNIX. La parte del sistema operativo que asigna recursos y controla procesos. La estrategia de diseño ha sido mantener el núcleo lo más pequeño posible y colocar el resto del sistema operativo UNIX en programas compilados y ejecutados por separado.

Número de bloque Discos y cintas están divididos en secciones (por lo general de 512 bytes de longitud, aunque puede ser mayor en algunos sistemas) numeradas para que el UNIX pueda seguir la pista de los datos del dispositivo. Los números son números de bloque.

Número de dispositivo Número asignado a un dispositivo en el momento de generar el sistema. Los números de dispositivo se listan con el mandato **ls** en el directorio **/dev.**

Número hexadecimal Número de base 16; compuesto por los dígitos hexadecimales del cero al nueve y de la A a la F. Vease la tabla siguiente.

Decimal	Hexadecimal	Decimal	Hexadecimal
1	1	14	E
2	2	15	F
3	3	16	10
4	4	17	11
5	5	20	14
6	6	31	1F
7	7	32	20
8	8	33	21
9	9	64	40
10	A	96	60
11	B	100	64
12	C	128	80
13	D	256	100

Número mayor (major device number) Número asignado a una clase de dispositivo tales como terminales, impresoras o unidades de disco. Si se usa la utilidad **ls** con la opción **-l** para listar el contenido del directorio **/dev**, se despliegan los números mayor y menor de todos los dispositivos.

Número menor (minor device number) Número asignado a un dispositivo específico de una clase de dispositivos, véase número mayor.

Número octal Número de base ocho; compuesto por los dígitos del cero al siete, ambos inclusive.

Omisión, por Seleccionar algo sin especificarlo de forma explícita. Cuando el mandato **ls** se utiliza sin argumentos, éste despliega, por omisión, una lista corta de los archivos existentes en el directorio de trabajo.

Ordenamiento Poner en un orden específico, por lo común alfabético o numérico.

Permiso de acceso Permiso para leer de un archivo, escribir en él o ejecutarlo. Por ejemplo, si se tiene permiso de acceso de escritura a un archivo, puede escribirse en éste.

Pie de página Parte de un formato que va en el extremo inferior de una página.

Proceso Medio por el cual UNIX ejecuta un programa.

Proceso preferente Cuando un programa se ejecuta de manera preferente, está ligado al terminal. A diferencia del proceso subordinado, en éste debe esperarse a que termine su ejecución para poder dar otro mandato al Shell).

Proceso subordinado Proceso que no se ejecuta de forma preferente, también se denomina proceso separado. Un proceso separado. Un proceso subordinado se inicia con una línea de mandato que termina con un signo &. No hay que esperar a que termine el proceso para digitar otros mandatos de Shell.

Programa Shell Proceso compuesto por mandatos Shell.

Rama En una estructura de árbol, ésta conecta los nodos, las hojas y la raíz.

Referencia ambigua a un archivo Referencia a un arc;hivo que no especifica necesariamente uno en particular, sino que puede servir para especificar un grupo de archivos. El Shell expande una referencia ambigua en una lista de nombres de archivos. Se utilizan caracteres especiales para representar caracteres individuales (?), cadenas de cero o más caracteres (*) y clases de caracteres ([]) dentro de una referencia ambigua.

Rutina recursiva Programa o subrutina que se llama a sí misma, ya sea en forma directa o indirecta.

Salida Informacion que un program envia al terminal o a otro archivo.

Salida estándar Archivo al que un programa puede enviar su salida. A menos que se indique al Shell lo contrario, la salida estándar se dirige al terminal.

Salida estándar con errores. Archivo al que un programa puede enviar su salida. Por lo comun, este dirige esta salida a la terminal.

Sangría Hablando de texto, el espacio en blanco que existe entre el margen y el comienzo de una linea.

Secuencia en ordenamiento de máquina Forma en que un computador ordenamiento y de otro tipo que relacionan listas de manera alfabetica. La mayor parte de los computadores utilizan ASCII.

Señal mensaje breve que UNIX envia a un proceso, sin tener en cuenta la entrada estándar del proceso.

Sesión Secuencia de eventos que ocurren desde que se comienza a usar un programa como el editor hasta que se termina, o desde que el usuario se da de alta hasta que se da de baja.

Shell Intérprete de mandatos de UNIX.

Shell de entrada El Shell que se utiliza al darse de alta por primera vez. Este Shell puede generar otros procesos que ejecutan otros Shells.

Spool Formación realizada para que algunosd elementos guarden su turno para llevar a cabo una accion. A menudo se usa con la utilidad **lpr** y con la impresora; **lpr** envia archivos a la impresora.

Subdirectorio Directorio localizado dentro de otro directorio.

Superusuario Usuario privilegiado que tiene acceso a cualquier cosa a lo que tenga acceso cualquier usuario, y más. El administrador del sistema debe estar capacitado para convertirse en su superusuario para habilitar nuevas cuentas, cambiar contraseñas y realizar otras funciones administrativas.

Swap Véase cambio (swap).

Termcap Abreviatura de terminal capability (capacidad del terminal). El archivo **termcap** contiene una lista de diferentes tipos de terminales y sus características. Los programas orientados a manejo de pantalla, como el ditor **vi**, utilizan este archivo.

tty Un terminal; tty es la abreviatura de teletypewriter.

Variable Nombre y valor asociados que se usan en un programa como, por ejemplo, el Shell.

* BIBLIOGRAFIA

- El entorno de programación UNIX
  Brian W. Kernighan, Rob Pike
  Prentice Hall HISPANOAMERICANA, MEXICO 1987

- Guía práctica para el sistema operativo UNIX
  Mark G. Sobell
  Addison-Wesley iberoamericana, MEXICO, 1987

- Revista Ciencia y Desarrollo
  noviembre, diciembre 1988 Vol XIV Num 83
  Tema : La era digital
  El sistema operativo Unix Paginas 169 - 178
  Autor: Guillermo Leavine

- Revista mensual UNIXWORLD
  editada por Mac Graw Hill
  L.C F.E.H.U 1990

- UNIX manual de referencia (Sistema V Versión 3)
  Stephen Coffin
  Osborne/Mc Graw Hill, INTERAMERICANA DE ESPAÑA 1989

- XENIX/UNIX User's Guide
  Altos Computer Systems

- XENIX System V Reference (C, M) Part One
  Altos Computer Systems

- XENIX System V Operations Guide
  Altos Computer Systems

- XENIX Commands Directory
  Altos Computer Systems

- XENIX User's Guide
  Altos Computer Systems