EVALUACION DEL PERSONAL DOCENTE

CURSO: SISTEMA OPERATIVO UNIX PARTE III
FECHA: 19 SEPT. AL 07 DE OCT. 1994

CONFERENCISTA	DOMINIO DEL TEMA	USO DE AYUDAS AUDIOVISUALES	COMUNICACION CON EL ASISTENTE	PUNTUALIDAD
ING. EDWIN NAVARRO				
ING. JESSICA BRISEÑO				
ING. JOSE A. CHAVEZ F.				

EVALUACION DE LA ENSEÑANZA

ORGANIZACION Y DESARROLO DEL CURSO	
GRADO DE PROFUNDIDAD LOGRADO EN EL CURSO	
ACTUALIZACION DEL CURSO	
APLICACION PRACTICA DEL CURSO	

EVALUACION DEL CURSO

CONCEPTO	CALIF.
CUMPLIMIENTO DE LOS OBJETIVOS DEL CURSO	
CONTINUIDAD EN LOS TEMAS	
CALIDAD DEL MATERIAL DIDACTICO UTILIZADO	

ESCALA DE EVALUACION: 1 A 10

1.- ¿LE AGRADO SU ESTANCIA EN LA DIVISION DE EDUCACION CONTINUA?

SI	NO

SI INDICA QUE "NO" DIGA PORQUE.

COORDINACION CURSOS DE COMPUTO
CENTRO DE INFORMACIÓN Y DOCUMENTACION

2.- MEDIO A TRAVES DEL CUAL SE ENTERO DEL CURSO:

PERIODICO EXCELSIOR		FOLLETO ANUAL		GACETA UNAM		OTRO MEDIO	
PERIODICO EL UNIVERSAL		FOLLETO DEL CURSO		REVISTAS TECNICAS			

3.- ¿QUE CAMBIOS SUGERIRIA AL CURSO PARA MEJORARLO?

4.- ¿RECOMENDARIA EL CURSO A OTRA(S) PERSONA(S)?

SI		NO	

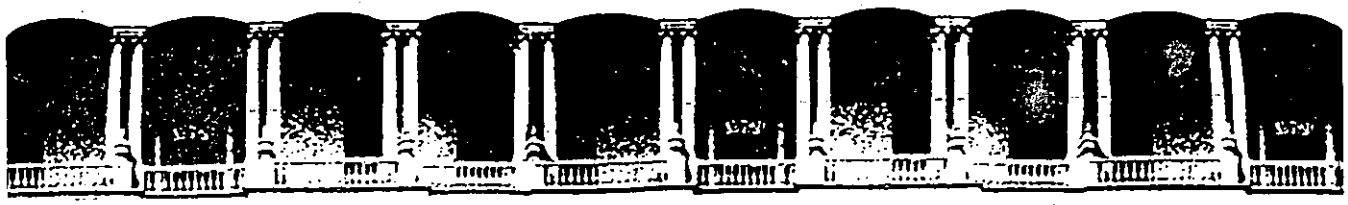
5.- ¿QUE CURSOS LE SERVIRIA QUE PROGRAMARA LA DIVISION DE EDUCACION CONTINUA.?

6.- OTRAS SUGERENCIAS:

7.- ¿EN QUE HORARIO LE SERIA CONVENIENTE SE IMPARTIERAN LOS CURSOS DE LA DIVISION DE EDUCACION CONTINUA? MARQUE EL HORARIO DE SU AGRADO

LUNES A VIERNES DE 16 A 20 HORAS	MARTES Y JUEVES DE 17 A 21 HS SABADO DE 10 A 14 HS.	OTRO
LUNES, MIERCOLES Y VIERNES DE 17 A 21 HORAS	VIERNES DE 17 A 21 HS. SABADOS DE 10 A 14 HS	

COORDINACION CURSOS DE COMPUTO
CENTRO DE INFORMACIÓN Y DOCUMENTACION



FACULTAD DE INGENIERIA U.N.A.M.
DIVISION DE EDUCACION CONTINUA

SISTEMA OPERATIVO UNIX - PARTE III-

INTRODUCCION

REDIRECCIONAMIENTO, FILTROS E INTERCONEXION DE COMANDOS

PROGRAMACION CON AWK

ING. JOSE ANTONIO CHAVEZ FLORES

SEPTIEMBRE 1994

Palacio de Minería Calle de Tacuba 5 Primer piso Deleg. Cuauhtémoc 06000 México, D.F. Tél.: 521-40-20 Apdo. Postal M-2285

Introducción

La importancia del establecimiento de estándares dentro del desarrollo de sistemas de computación es indiscutible. En los últimos años, la tecnología de hardware ha evolucionado notablemente, a tal grado que en 1993 podemos encontrar microcomputadoras con velocidades de procesamiento de datos comparables a las de las poderosas mainframes y estaciones de trabajo que muchas veces sobrepasan las capacidades de las minicomputadoras. Este hecho ha provocado que la necesidad de intercambio de información sea más grande, lo que ha ocasionado el surgimiento de las famosas redes de computadoras, que permiten la interconexión remota de computadoras de diferentes tecnologías.

Dentro de este panorama, es necesario establecer una serie de estandáres que permitan el desarrollo de sistemas portables, así como el intercambio de información. Al conjunto de dichos estándares se les conoce actualmente como **sistemas abiertos** (*Open Systems*). Un sistema abierto es aquel que ha sido adoptado en diferentes arquitecturas de hardware y software debido a que sus especificaciones de diseño son públicas y por lo tanto se ha estandarizado. Se pueden mencionar a productos como **TCP/IP** (*Transmission Control Protocol/Internet Protocol*) en el área de protocolos de comunicación entre computadoras y a **X-Window System** dentro de las interfases gráficas de usuario.

En el campo de los sistemas operativos, también es necesario establecer un estándar que permita a máquinas con diferentes arquitecturas utilizar un mismo sistema operativo, para que de esta forma todos los desarrollos de aplicaciones se den en un mismo ambiente y se de paso a sistemas totalmente portables. El sistema operativo UNIX se ha convertido en uno de los entornos de programación más populares y utilizados del mundo. Actualmente, miles de computadoras, que van desde las microcomputadoras hasta las supercomputadoras, utilizan UNIX. Se puede afirmar que UNIX es el sistema operativo que se ha establecido como sistema abierto hasta el momento.

¿A qué se debe el éxito del sistema operativo UNIX? Además de que se cuenta con las especificaciones de diseño de UNIX, podemos distinguir dos razones principales por las cuales se ha hecho tan popular. Primero, esta escrito en lenguaje C, lo que lo hace portátil; segundo, es un buen sistema operativo, su ambiente de programación es de extraordinaria riqueza y productividad.

La eficiencia de UNIX radica en su enfoque de la programación, que constituye una filosofía de cómo utilizar la computadora y a su consistencia debida a que el sistema operativo fue diseñado por un grupo muy pequeño de personas.

1. Historia de UNIX

Entre 1965 y 1969, los Laboratorios Bell de AT&T participaron, junto con General Electric y el Instituto Tecnológico de Massachusetts, en el desarrollo del sistema Multics. El sistema fue originalmente diseñado para operar en una computadora GE-645; sin embargo el sistema era demásiado grande y complejo, por lo que los Laboratorios Bell abandonaron el proyecto en 1969. Sin embargo, el grupo de investigadores de los Laboratorios Bell que participo en el proyecto original, se propuso crear un sistema operativo que fuera lo suficientemente cómodo y rápido y que además facilitara la investigación y desarrollo de programas. La primera implementación de UNIX se hizo en una PDP-7 de DEC y se escribió en lenguaje ensamblador. Participaron Ken Thompson, Rudd Canaday, Doug McIlroy, Joe Ossanna y Dennis Ritchie.

La popularidad de UNIX se extendió dentro de los Laboratorios Bell y poco tiempo después la era conocido por la mayoría de los investigadores de dichos Laboratorios. Thompson propuso la posibilidad de transportar el nuevo sistema operativo a otras máquinas; este trabajo requería de rehacer el 90% del trabajo realizado y así poder justificar la adquisición de una máquina DEC PDP 11-20. En 1970 Thompson con ayuda de Ritchie trasladó UNIX a la PDP 11.

Dennis Ritchie tuvo un papel muy importante en la codificación del núcleo de UNIX en lenguaje C, en 1972. Esto ayudó a hacer más portátil y comprensible al sistema. El código de máquina del sistema resultó mayor que la versión en lenguaje ensamblador, pero parte del aumento se debió a la adición del apoyo a la multiprogramación y a la posibilidad de compartir procedimientos reentrantes.

El sistema UNIX captó inmediatamente la atención de los investigadores de la AT&T quienes comenzaron a utilizar máquinas PDP-11 con UNIX para el desarrollo de sistemas telefónicos.

En 1975, UNIX se había hecho muy popular en las universidades, ya que se instaló en ellas con fines educativos. Esto dio lugar a una serie de innovaciones e implementaciones de UNIX dentro de las cuales se encuentra la realizada en la Universidad de California en Berkeley. Esta versión denominada **BSD** (*Berkeley Software Distribution*) fue la más difundida y utilizada dentro de la comunidad universitaria de Estados Unidos. La compatibilidad entre las versiones AT&T y BSD sigue siendo hasta cierto punto cuestionable. Las versiones BSD son equiparables con las versiones AT&T, ya que ambas incorporan las mejores innovaciones del otro sistema a sus propias versiones.

El primer sistema UNIX fue la versión seis que constituyo un desarrollo interno de los Laboratorios Bell en el año 1977. La versión oficial de UNIX fue la séptima, liberada en 1979. Las versiones más populares fueron la sexta y la séptima. Siguiendo una reorganización interna del soporte del sistema UNIX, AT&T cambió su numeración a Sistema III y Sistema V; sin embargo el Sistema V dominó ampliamente al Sistema III. El Sistema IV fue utilizado internamente en los Laboratorios Bell, pero se consideró un producto de transición que nunca fue soportado

públicamente. A finales de los ochenta, AT&T normalizó el nombre de Sistema V versión 2 y Sistema V versión 3 (SVR2 y SVR3).

En 1981, Microsoft desarrollo XENIX, que constituye la versión de UNIX para microcomputadoras con microprocesadores de 16 bits.

En 1980 DEC (Digital Equipment Corporation) sacó al mercado su versión de UNIX denominada ULTRIX, la cual incorpora todas las facilidades de la versión 4.2 BSD e incorpora mejoras en el área de comunicaciones. En 1983 apareció en el mercado la versión 4.3 de BSD. Sun Microsystems desarrolló su versión de UNIX denominada SunOS basada en la versión 4.2 BSD, la cual soporta facilidades para ambientes gráficos de ventanas e interfase de ratón en un sistema interconectado de estaciones de trabajo.

Las versiones descendientes de BSD y AT&T están siendo constantemente mejoradas; permitiendo integrar las diferentes versiones en una sola. Se espera que las diferentes versiones converjan en una versión única de UNIX que pueda funcionar en cualquier arquitectura de computadora.

2. Características y componentes de UNIX

UNIX es un sistema operativo con características muy especiales, lo que le ha valido el reconocimiento por parte de los diseñadores de sistemas. Dentro de sus características principales, podemos mencionar las siguientes:

- Sistema **multiusuario** y **multitarea.**

- Las especificaciones de diseño están disponibles públicamente, lo cual hace que se adapte a exigencias particulares.

- Esta escrito en un lenguaje de alto nivel, lo que lo hace portátil.

- Enfoque de programación.

- Redireccionamiento, filtros e interconexiones.

- Sistema de archivos sencillo y eficiente.

- Un manejo de archivos consistente.

- La interfase con los dispositivos periféricos se maneja igual que un archivo.

- Esconde la arquitectura del hardware que lo utiliza.

- Es fácil de utilizar.

Los componentes o la arquitectura de alto nivel de UNIX se pueden representar por el diagrama de la figura 2. Si vemos el sistema como un conjunto de capas, el sistema operativo propiamente dicho conforma la capa más interna, esta parte del sistema operativo es la que se comunica directamente con el hardware y se le conoce comúnmente como **kernel** o **núcleo.** Entre las funciones del núcleo se pueden mencionar las siguientes: planificación de tareas, administración de recursos del sistema, administración de procesos y manejo de memoria.

En la siguiente capa se encuentran las utilerías y programas de aplicación vinculados con el sistema que se encargan de ejecutar una variedad de rutinas y funciones especiales de mantenimiento del sistema. Estas utilerías se comunican con el kernel por medio de una interfase desarrollada en lenguaje C que se conoce como **llamadas al sistema.** Muchas de las utilerías y programas forman parte de la configuración estándar de UNIX y son conocidos comúnmente como comandos.

Uno de los programas de aplicación más importantes es el intérprete de comandos o **shell.** Este programa se ejecuta inmediatamente después de que se abre una sesión de UNIX, es un proceso que interpreta los comandos que teclea el usuario y ejecuta las acciones asociadas con dichos comandos; de esta forma se lleva a cabo la comunicación entre el usuario y el sistema operativo.

Existen varios tipos de shell entre los que se encuentran *Bourne shell, C-shell, Reduced shell, Korn shell* y *Visual shell.* El usuario puede desarrollar su propio interprete de comandos para que sea este el que lo comunique con el núcleo.

En la capa superior de la arquitectura de UNIX se encuentran los programas de aplicación que no están comprendidos dentro de la configuración estándar de UNIX, es decir programas creados por los usuarios.

3. Estandarización de la interfase de UNIX

Actualmente se están desarrollando una gran cantidad de aplicaciones para ambientes UNIX, aplicaciones que van desde editores, hojas de cálculo, manejadores de bases de datos distribuidas, software de CAD/CAM, hasta aplicaciones tan especializadas como simuladores médicos y mecánicos, software gráfico de red, etc. Esto ha originado que los desarrolladores de software tengan necesidad de que la interfase para accesar los servicios del hardware por medio del sistema operativo sea estándar y de esta forma poder desarrollar más fácilmente sistemas portables de una plataforma de hardware a otra.

Los servicios que proporciona el kernel de UNIX se pueden accesar a través de una interfase en lenguaje C formada por funciones denominadas **llamadas al sistema**, dichas funciones deben de ser definidas de una forma semejante en cada una de las implementaciones de UNIX.

El estándar conocido como **IEEE 1003.1 POSIX** describe cada llamada al sistema en detalle, incluyendo una sinopsis, la descripción de la llamada, de los parámetros que recibe, los valores de regreso y los posibles errores que se generan.

El estándar de POSIX incluye llamadas al sistema para manejo de señales, intercomunicación de procesos, manejo del sistema de archivos, manejo de dispositivos, administración de la memoria, comunicaciones, etc.

Actualmente, como estrategia de migración a UNIX, muchos sistemas propietarios incluyen POSIX en su configuración base.

Capítulo 2

Redireccionamiento, filtros e interconexión de comandos

Una de las características principales de UNIX es su enfoque de programación. Dentro de este enfoque tenemos lo que se conoce como **interconexiones** o **pipes**, que permiten dirigir la salida de un comando como entrada de otro, con lo cual podemos obtener por medio de comandos aparentemente sencillos resultados que en otros sistemas se realizan con comandos muy especializados.

Algunos de los comandos que ayudan a realizar esta interconexión, son los llamados **filtros**, que son comandos sencillos que leen alguna entrada, realizan una serie de transformaciones sobre esta y generan una salida sin modificar la entrada original de datos. En este capítulo se discutirán los filtros más utilizados, así como también la forma de utilizarlos en interconexiones.

1. Redireccionamiento de entrada/salida

La mayoría de los comandos y utilerías de UNIX toman el flujo de datos de entrada, por omisión, de la **entrada estándar** (definida como el teclado) y producen una salida de datos hacia la **salida estándar** (definida como el monitor); sin embargo, tanto la entrada de datos como la salida se pueden cambiar, de manera que los comandos puedan tomar datos de entrada de un archivo y su salida se pueda escribir a un archivo o sea tomada como entrada a otro comando.

Cuando se modifica la salida estándar de datos se dice que se esta haciendo un **redireccionamiento de salida.** El redireccionamiento puede ser dirigido hacia un archivo con ayuda del operador >. El ejemplo siguiente, muestra como se realiza el redireccionamiento de salida:

```
% ls -la > listado
%
```

El comando ls -la, normalmente va ha desplegar un listado completo de todos los archivos que se encuentran en el directorio de trabajo; cuando se utiliza el operador de redireccionamiento, la salida que produce el comando se va ha almacenar en el archivo denominado listado. En la pantalla de la terminal no aparecerá nada, ya que el flujo de datos de salida se mando hacia el archivo:

```
% cat listado
drwx------  3 alu01      33792 Sep  5  1990 .
drwxr-xr-x 64 root        5120 May  1  1990 ..
drwx------  1 alu01      33792 Sep  5  1990 .cshrc
drwx------  1 alu01       5120 Sep  5  1990 .exrc
drwx------  1 alu01      33792 Sep  5  1990 .login
-rwx------  1 alu01       5120 Sep  1  1990 .logout
-rwx------  1 alu01      51200 Sep  1  1990 .profile
drwxr-xr-x  2 alu01      33792 Apr  5  11:35 bin
-rwxr-xr-x  1 alu01       5120 Apr  1  19:20 tarea.c
-rwxr-xr-x  2 alu01      51200 Apr  1  15:19 tarea_so
-rwxr-xr-x  2 alu01      51200 Apr  1  15:19 tarea_so.ln
```

El archivo será creado en caso de que no exista; si ya existe, su contenido será reemplazado. De esta forma, es posible con ayuda del comando cat almacenar el contenido de varios archivos en uno sólo:

```
% cat tarea1 tarea2 tarea3 > tareas
%
```

El operador de redireccionamiento >> funciona de manera semejante a como lo hace >, excepto que >> no reemplaza el contenido del archivo a donde se redirecciona la salida, sino que conserva su contenido y además anexa al final del archivo la salida que se esta redireccionando.

De esta forma, el comando:

```
% cat tarea1 tarea2 tarea3 >> tareas
%
```

crea el archivo tareas en caso de que no exista, de lo contrario se va ha anexar el contenido de los archivos tarea1, tarea2 y tarea3 al final del contenido del archivo tareas.

El **redireccionamiento de entrada** se da cuando se sustituye la entrada estándar por un archivo. El redireccionamiento de entrada utiliza el operador <. Por ejemplo, para redireccionar la entrada del comando cat:

```
% cat < tarea1.c
/*  Tarea No. 1

     Programa que manda un letrero de "hola mundo"
*/

#include <stdio.h>

main() {
     printf("Hola mundo\n");
}
%
```

El comando anterior toma su entrada del archivo tareas y despliega en la salida estándar su contenido. Supongase que se tiene un programa llamado nomina al cual se le proporcionan como entrada una lista de empleados y genera como salida un reporte en pantalla. Se podrían hacer una serie de manipulaciones para que la lista de empleados se proporcionará por medio de un archivo y el reporte quedará almacenado en otro:

```
% nomina < empleados > reporte
%
```

El archivo empleados contiene una lista de empleados y en el archivo reporte se almacenará el reporte que normalmente se produce en pantalla al ejecutar el programa nomina.

2. Filtros

Un filtro en UNIX es un programa o comando que recibe un flujo de datos de entrada, realiza una transformación, manipulación o selección de estos datos y genera una salida sin alterar la entrada original.

El comando **tail** es un filtro que despliega las últimas diez líneas del flujo de datos de entrada. La siguiente línea de comando:

```
% tail tareas
```

desplegará las últimas diez líneas del archivo tareas. El comando tail también permite que se le indique cuantas líneas se desea desplegar. Así, para desplegar las últimas 20 líneas del archivo tareas, se deberá ejecutar el siguiente comando:

```
% tail -20 tareas
```

También se puede utilizar tail para desplegar el contenido de un archivo a partir de una línea específica:

```
% tail +10 tareas
```

desplegará el archivo tareas a partir de la línea 10.

El comando **head** funciona de una forma semejante a como lo hace tail, solamente que head despliega las diez primeras líneas del flujo de datos de entrada:

```
% head tareas
```

También se puede indicar cuantas líneas se desean desplegar:

```
% head -20 tareas
```

muestra las 20 primeras líneas del archivo tareas.

Otro filtro interesante es **wc** (*word counter*), el cuál permite hacer un conteo de las palabras, líneas y caracteres de un flujo de datos proporcionado como entrada. El siguiente ejemplo muestra el funcionamiento de wc:

```
% wc tarea1.c nomina
10    20    130 tarea1.c
5     40    153 nomina
15    60    283 total
%
```

Por omisión, wc cuenta el número de líneas, palabras y caracteres en el flujo de datos de entrada; sin embargo, se pueden especificar las siguientes opciones, o una combinación de ellas:

l	cuenta líneas
w	cuenta palabras
c	cuenta caracteres

por ejemplo, el siguiente comando:

```
% wc -lw tarea1.c
10    20 tarea1.c
%
```

da como resultado el número de líneas y palabras en el archivo tarea1.c.

El comando **sort** permite ordenar registros de un flujo de datos de entrada. La clasificación se hace tomando como base el orden lexicográfico de cada uno de los caracteres en un registro; sin embargo, se puede llevar a cabo el ordenamiento tomando un campo especifico como llave de ordenamiento. Los registros de entrada son vistos como un conjunto de campos separados por espacios en blanco o tabuladores. Por ejemplo, supongase que se desea ordenar el archivo nombres, cuyo contenido se muestra en la figura 3.1, tomando como llave de ordenamiento el primer apellido; para este caso el comando a ejecutar sería:

```
% sort +1 nombres
Aaron Arcos Tapia
Norberto Arrieta Marquez
Beatriz Barrera Hernandez
Jessica Briseño Cortes
Antonio Chavez Flores
Gabriela Magallanes Gonzalez
Gerardo Meza Gil
Edwin Navarro Pliego
%
```

El parámetro +1 especifica el campo que se tomará como llave del

ARCHIVO: *nombres*

```
Antonio Chavez Flores
Norberto Arrieta Marquez
Aaron Arcos Tapia
Jessica Briseño Cortez
Gabriela Magallanes Gonzalez
Gerardo Meza Gil
Edwin Navarro Pliego
Beatriz Barrera Hernandez
```

ARCHIVO: *frutas*

```
manzana          4500
naranja          1500
pera             6000
platano           800
melon            2000
sandia            900
guayaba          1200
fresa            5000
mango            1100
```

Fig. 3.1. Contenido de los archivos nombres y frutas

ordenamiento; el primer campo se indica como +0.

Otras opciones de sort se muestran en la siguiente tabla

OPCION	FUNCION
f	Considera letras mayúsculas y minúsculas por igual.
n	Toma el campo a clasificar como un valor numérico.
r	El ordenamiento se realiza en orden inverso.
t	Especifica un separador de campo.
u	Elimina registros repetidos.
m	Ordena por el método de la mezcla archivos previamente clasificados.

El siguiente ejemplo:

```
% sort +1 frutas
mango           1100
guayaba         1200
naranja         1500
melon           2000
manzana         4500
fresa           5000
pera            6000
platano          800
sandia           900
%
```

no produce el resultado esperado, puesto que el ordenamiento se hace tomando el orden lexicográfico de los caracteres que componen un campo, de tal forma, que el segundo campo es tomado simplemente como una secuencia de caracteres y no como un valor numérico. Para que el resultado sea correcto se debe especificar un ordenamiento numérico tomando como llave el segundo campo:

```
% sort +1n frutas

platano          800
sandia           900
mango           1100
guayaba         1200
naranja         1500
melon           2000
manzana         4500
fresa           5000
pera            6000
%
```

3. Interconexión de comandos

Al utilizar redireccionamiento de entrada/salida se vio que se pueden crear archivos que contengan la salida de un comando; de esta forma, se podría tomar dicho archivo como entrada a otro comando. Por ejemplo, supongase que se desea obtener en pantalla el desplegado de los archivos que existen en el directorio /bin ordenados por su tamaño. Recordando que con la opción **l** del comando ls el cuarto campo que se despliega indica el tamaño en bytes, se puede, con los siguientes comandos, obtener el resultado deseado:

```
% ls -la /bin > temporal
% sort +3n temporal
```

Sin embargo, el archivo creado no tiene ninguna utilidad después de que se ha obtenido el resultado. Si el usuario del sistema realizará algunas otras aplicaciones de este tipo, tendría que estar creando archivos temporales, los cuales resultarían innecesarios. El sistema operativo UNIX nos da la facilidad de ahorrar la creación de los archivos temporales, redireccionando o conectando la salida de un programa como entrada a otro mediante una interconexión; de esta forma, para obtener el resultado anterior bastaría con ejecutar el siguiente comando:

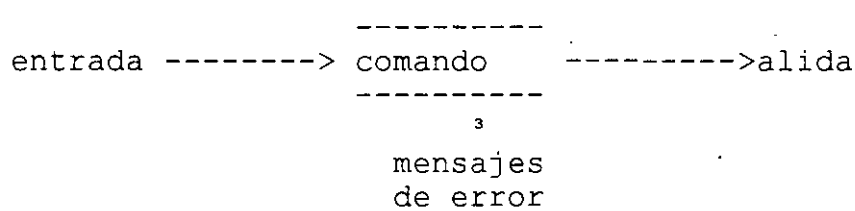
```
% ls -la /bin | sort +3n
```

Todos los programas que reciben como entrada un flujo de datos ya sea de la entrada estándar o de un archivo, lo pueden hacer de otro programa a través de una interconexión y la salida de ellos puede pasar como entrada a otro comando o redireccionarse hacia un archivo. Este tipo de programas deben operar correctamente y generalmente, el esquema para invocarlos es el siguiente:

comando [argumentos] [lista de archivos]

donde los corchetes indican que se trata de parámetros opcionales. De esta forma, el flujo de datos de entrada para un comando puede provenir de un conjunto de archivos, de la entrada estándar o de una interconexión; la salida es generada, por omisión, hacia la salida estándar, para que de esta forma se pueda redireccionar hacia un archivo o hacia una interconexión.

Los mensajes de error de los comandos no se mezclan con la salida normal, sino que son enviados a la **salida de errores** (definida como el monitor); de esta forma, los mensajes de error no se pierden al utilizar interconexiones. El siguiente diagrama ilustra el diseño de este tipo de comando:



Cuando se interconectan comandos, la única salida que se puede manipular, ya sea para almacenarse en un archivo o para desplegarla en pantalla, es la del último comando interconectado; esto se muestra en la figura 3.2.

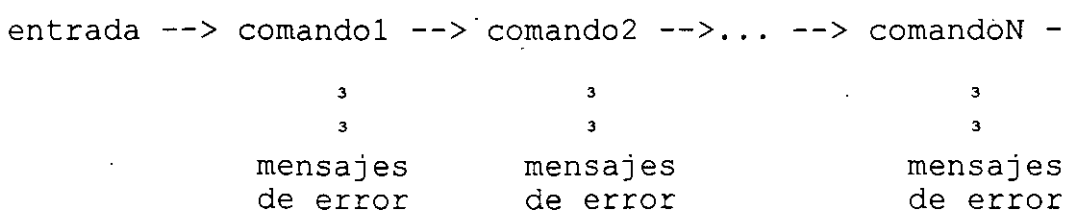


Fig. 3.2. Interconexión de comandos.

4. Expresiones regulares con grep y egrep

Otro filtro de gran utilidad es **grep** (*global regular expression printer*), el cual busca la ocurrencia de un patrón en el flujo de datos de entrada y despliega las líneas donde encuentra dicho patrón. Por ejemplo, para saber si el usuario con clave curso101 se encuentra en sesión, se podría teclear el siguiente comando:

```
% who | grep curso101
curso101    tty01     Feb 10 16:40
%
```

El patrón que se especifica en grep, se conoce como expresión regular. Una expresión regular es un patrón que representa o describe a un conjunto de cadenas. En el ejemplo anterior, la expresión regular era sencilla; sin embargo, las expresiones regulares se especifican utilizando caracteres que tienen significado especial dentro de la misma expresión regular. A estos caracteres se les denomina metacaracteres, por ejemplo: **^** indica inicio de línea y **$** fin de línea.

Cuando se utilizan metacaracteres en la expresión regular, esta debe encerrarse entre apóstrofes. Por ejemplo, para listar todos los directorios del directorio de trabajo, se utilizaría el siguiente comando:

```
% ls -la | grep '^d'
```

recuerde, que el comando ls con la opción **l**, proporciona varios campos de información para los archivos y que el primer carácter del primer campo indica el tipo de archivo del que se trata (**d** para directorios, - para archivos ordinarios).

El metacaracter **.** se utiliza para representar a un carácter cualquiera, ***** indica cero o más repeticiones del carácter anterior (más adelante se vera que se utiliza también para representar cero o más ocurrencias de una cadena), de tal forma que la expresión regular **a.*b** representa a cadenas que comienzan con el carácter **a** seguidas de una secuencia de caracteres (no importando cuales, incluyendo la cadena vacía) y terminan con **b**.

Se ha mencionado que el carácter . representa a cualquier carácter y que para representar a uno en especial basta con representarlo tal como es en la expresión. Si se quiere especificar un subconjunto de caracteres se especifica este rango entre corchetes, por ejemplo, **[a-z]** representa a cualquier letra minúscula, **[a-zA-Z]** a cualquier letra mayúscula o minúscula.

Supongase que se desea obtener una lista ordenada, del archivo nombres, de las personas cuyo primer apellido comience con las tres primeras letras del alfabeto; la lista deberá estar ordenada por apellido paterno. En este caso, el comando adecuado sería el siguiente:

```
% grep '^[abcABC]' nombres | sort +1
Aaron Arcos Tapia
Beatriz Barrera Hernandez
Jessica Briseño Cortes
Antonio Chavez Flores
%
```

El comando grep acepta las opciones que se muestran en la tabla 3.2.

Existen otros dos comandos cuyo funcionamiento es muy similar a grep y también utilizan expresiones regulares: **egrep** y **fgrep**. El primero de ellos utiliza expresiones regulares verdaderas, con algunos metacaracteres adicionales y el segundo comando no utiliza metacaracteres, pero con la opción -f se pueden buscar varios patrones fijos simultáneamente en un mismo archivo.

OPCION	FUNCION
c	produce como salida el número de líneas en donde fue encontrado el patrón.
f	especifica un archivo del cual se lee el patrón.
n	genera como salida la línea en donde se encontró la ocurrencia del patrón, precedida por su número de línea.
s	No genera ninguna salida, salvo que se trate de mensajes de error.
v	Despliega las líneas que no se ajustan a la expresión regular proporcionada.

Tabla 3.2. Opciones de grep.

Las expresiones regulares que se le proporcionan a egrep, pueden utilizar como metacaracteres a los paréntesis para agrupar expresiones, por ejemplo: **(ab)*** representa a la cadena vacía y a cadenas que tienen la secuencia ab, como la cadena abababababab. El comando egrep también utiliza al metacaracter | como operador or; por ejemplo la expresión **(ab|xy)*** representa a la cadena vacía o a una cadena con secuencias de ab o xy, como la cadena xyxyabxyabab.

Suponga que se desea buscar los archivos o directorios en el directorio de trabajo que tienen protecciones de lectura y escritura o sólo lectura para todos los usuarios, el comando utilizado sería el siguiente:

```
% ls -la | egrep '^[-d].....(rw-|r--)'
```

Otro conjunto de metacaracteres importantes son + y ?, el primero indica una o más ocurrencias del carácter o cadena anteriores (no incluye a la cadena vacía); el segundo indica una ocurrencia o la cadena vacía.

La tabla resume los metacarateres para grep y egrep.

METACARACTER	SIGNIFICADO
c	Cualquier carácter que no sea especial se representa a sí mismo.
^	Inicio de línea.
$	Fin de línea.
*	Cero o más repeticiones del carácter o cadena anteriores.
+	Una o más repeticiones del carácter o cadena anteriores, no incluye a la cadena vacía (solamente en egrep).
?	Cero o una repetición del carácter o cadena anteriores (solamente en egrep).
[]	Representa al rango de caracteres encerrado en los corchetes, por ejemplo a-z; ^a-z es la negación del rango.
exp1|exp2	Representa a la expresión regular uno o a la expresión regular dos.

Tabla 3.3. Metacaracteres para grep y egrep.

Capítulo 3

Programación con AWK

Uno de los filtros más importantes es **awk** (el nombre se debe a las iniciales de sus creadores: Alfred V. **A**ho, Brian **W**. Kernighan y Peter J. **W**einberger), un localizador de patrones y lenguaje de programación que examina un flujo de datos de entrada y compara cada línea con el conjunto de patrones especificados. Para cada patrón, se ejecuta una serie de acciones indicadas a través de un lenguaje de programación.

Los patrones pueden ser expresiones regulares como las utilizadas en grep y egrep. El programa consiste de una serie de instrucciones cuya sintaxis es muy parecida a la del lenguaje C. El programa puede ser especificado en la línea de comandos, o bien por medio de un archivo indicandolo con la opción ***f*** en la línea de comandos:

```
awk programa [lista_archivos]

awk -f archivo_de_programa [lista_archivos]
```

Cuando el programa se especifica en la línea de comandos, este debe encerrarse entre apóstrofes.

Un programa de awk es una secuencia de patrones asociados con una serie de acciones:

```
patrón { acciones }
patrón { acciones }
...
```

El patrón selecciona los registros o líneas para los cuales se van a ejecutar las acciones, si dicho patrón no se especifica, se seleccionan todos los registros del flujo de datos de entrada; por otra parte, si no se indica acción alguna, se imprimen todos los caracteres de las lineas seleccionadas en el patrón. El programa de awk es procesado para cada una de las líneas de entrada.

Al igual que sort, awk toma el flujo de datos de entrada como una secuencia de registros divididos en campos e identifica al primer campo como $1, al segundo como $2 y así sucesivamente. Por otra parte, $0 identifica a todo el registro. El separador de campos por omisión es el blanco.

Cuando se utilizan expresiones regulares como patrones en awk, estas se encierran entre diagonales; por ejemplo, si se desea listar los nombres de los directorios en el directorio de trabajo actual, se podría ejecutar el siguiente comando:

```
% ls -la | awk '/^d/ { print $8 }'
```

La acción indicada, es la de imprimir el campo ocho (en este caso el nombre del directorio) de cada una de las líneas seleccionadas con la expresión regular.

Para el archivo nombres, mencionado en el capítulo 3, si se quiere obtener un listado de los nombres comenzando con sus apellidos, se podría teclear el siguiente comando:

```
% awk '{ print $2,$3,$1 }' nombres
Chavez Flores Antonio
Arrieta Marquez Norberto
Arcos Tapia Aaron
Briseño Cortez Jessica
Magallanes Gonzalez Gabriela
Meza Gil Gerardo
Navarro Pliego Edwin
Barrera Hernandez Beatriz
%
```

La instrucción print causa que se haga una copia a la salida de los parámetros listados, las comas en print son reemplazadas a la salida por el separador de campos de salida. Cada vez que se manda una instrucción print, se manda a la salida el carácter de fin de línea. La salida de print puede ser dirigida a múltiples archivos; por ejemplo, se quieren mandar los apellidos del archivo nombres al archivo x y los nombres al archivo y, el comando a ejecutar sería:

```
% awk '{ print $2,$3 > "x"; print $1 > "y" }' nombres
```

Para separar instrucciones en una misma línea se utiliza ;. En un programa que este almacenado en un archivo cada instrucción se puede colocar en una línea, sin necesidad de separarlas con ;. En el ejemplo anterior se puede utilizar **>>** en lugar de **>** para producir el mismo resultado que cuando se hace redireccionamiento de salida.

Para tener un mejor control del formato que se da a la salida, se puede utilizar la instrucción **printf**, la cual tiene la siguiente sintaxis:

printf cadena_formato, arg_1, arg_2, ..., arg_n

La cadena de formato contiene caracteres ordinarios, que son copiados a la salida, y especificaciones de conversión, cada una de las cuales causa la conversión de los siguientes argumentos sucesivos de printf. Cada una de estas especificaciones comienzan con **%** y terminan con uno de los caracteres mostrados en la tabla 5.1.

CARACTER	FORMA EN LA QUE ES IMPRESO EL ARGUMENTO
d	Número decimal.

o	Número en formato octal.
x	Número en formato hexadecimal.
c	Caracter.
s	Cadena de caracteres.
f	Número con parte fraccionaria.

Tabla 3.1. Conversiones para printf

Entre el **%** y el caracter de conversión puede aparecer, en orden:

- Un signo menos, que indica especificación a la izquierda del argumento convertido.

- Un número que indica el ancho mínimo del campo.

- Un punto, que separa el ancho de campo de la precisión.

- Un número que indica el número de dígitos después del punto decimal para un valor numérico, o el número máximo de una cadena de caracteres.

Por ejemplo, el siguiente programa:

```
% awk '{printf"%3d %-10s %-10s %-10s\n",NR,$1,$2,$3}'
nombres
  1 Antonio    Chavez     Flores
  2 Norberto   Arrieta    Marquez
  3 Aaron      Arcos      Tapia
  4 Jessica    Briseño    Cortez
  5 Gabriela   Magallanes Gonzalez
  6 Gerardo    Meza       Gil
  7 Edwin      Navarro    Pliego
%
```

da formato a cada uno de los campos de salida. El printf no manda al flujo de salida el caracter de fin de línea, este se debe especificar en la cadena de control y se representa como **\n**. En el ejemplo anterior, se manda a la salida de datos la variable predefinida **NR**, la cual almacena el número de registro o línea

que se procesa en ese momento. Cada vez que awk procesa una línea diferente NR cambia su valor. Otras variables predefinidas en awk se muestran en la tabla 5.2.

VARIABLE	DESCRIPCION
NF	Número de campos de la línea de entrada.
FS	Caracter separador de campos.
RS	Caracter separador de líneas de entrada (por omisión es el caracter de fin de línea).
NR	Número de la línea de entrada actual.
OFS	Caracter separador de campos de salida.
FILENAME	Nombre del archivo de entrada actual.

Tabla 5.2. Variables predefinidas en awk.

Existen dos patrones especiales en awk, el primero de ellos es BEGIN, con el cual se especifican las acciones que se deberán realizar antes de que se comience a procesar la primera línea de entrada; el otro es END que especifica las acciones que se realizan después de haber leído la última línea de entrada. El patrón BEGIN es muy utilizado para hacer inicialización de variables.

Para ilustrar los patrones BEGIN y END, consideremos los siguientes ejemplos; primero, se obtendrá el número de líneas procesadas:

```
% awk 'END { printf "\t%d\n", NR }' nombres
      6
%
```

el resultado anterior se pudo haber obtenido también con el comando wc con la opción 1, tomando como entrada un sólo archivo.

El segundo ejemplo consiste en obtener las claves válidas en el sistema, para ello se deberá tener el siguiente antecedente: existe un archivo llamado /etc/passwd en donde se registran las claves del sistema, dicho archivo contiene un registro con información para cada una de las claves, el registro consiste de varios campos separados por el caracter :, el significado de los campos se muestra en el siguiente registro de ejemplo:

```
 curso101:15eT9M0Ggrxfs:310:15 clave para cursos:/usr/users/curso101:/bin/csh
   ³             ³   ³   ³               ³                 ³          ³
Clave ÄÙ             ³   ³   ³               ³                 ³          ³
Contraseña encriptadaÄÙ  ³   ³               ³                 ³          ³
Identificador de usuario ÄÙ  ³               ³                 ³          ³
Identificador de grupo ÄÄÄÄÄÄÄÙ               ³                 ³          ³
Comentario ÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÙ                 ³          ³
Directorio de HOME ÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÙ          ³
ShellÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÙ
```

Por lo tanto, para obtener el resultado deseado, se daría el siguiente comando:

```
% awk 'BEGIN { FS=":" } { print $1 }' /etc/passwd
```

1. Operaciones aritméticas

El lenguaje de programación de awk, emplea un conjunto de operadores para llevar a cabo operaciones aritméticas entre variables. Los operadores aritméticos son los siguientes:

+	Suma
-	Resta
*	Multiplicación
/	División
%	Residuo

En awk se manejan variables de tipo numérico y cadena; sin embargo, no es necesario definirlas, ya que se definen al momento de utilizarse. Tampoco es necesario inicializarlas, ya que por omisión, las variables numéricas se inicializan en cero y las tipo cadena lo hacen con la cadena vacía. Dependerá del contexto tratar a una variable como un número o como una cadena, en casos ambiguos, el valor de cadena se utiliza a menos que los operandos sean numéricos. Existen una serie de funciones para manipulación de variables, constantes numéricas o cadenas, estas se muestran en la tabla 5.3.

FUNCION	DESCRIPCION
cos(val)	Coseno de val.
sin(val)	Seno de val.
exp(val)	Exponencial de val.
int(val)	Obtiene la parte entera de val.
log(val)	Logaritmo natural de val.
length(cad)	Longitud de la cadena cad.
substr(cad,m,n)	Obtiene una subcadena de m caracteres de la cadena cad, comenzando en la posición n.
index(cad1,cad2)	Devuelve la posición a partir de la que se encuentra cad2 en cad1, o cero si no se encuentra.
sprintf(f,e1,...)	Devuelve una cadena con el formato especificado en la cadena f, tomando los argumentos e1, ...

Tabla 5.3. Funciones predefinidas en awk

Consideremos el ejemplo de un programa cuyo funcionamiento es igual al de wc cuando se le proporciona un flujo de datos de entrada proveniente de la entrada estándar o de un sólo archivo (más adelante generalizaremos el ejemplo para cuando se toman varios archivos de entrada). El programa se muestra en la siguiente figura .

```
{ caracteres = caracteres + length($0) + 1
  palabras = palabras + NF
}
END {
  printf "\t%d\t%d\t%d\n", NR, palabras,caracteres
}
```

Fig. 3.1. Programa que cuenta líneas, palabras y caracteres.

Las operaciones de asignación como la siguiente:

palabras = palabras + NF

en las cuales se modifica una variable con una operación sobre la misma, se pueden escribir en forma compacta de la siguiente forma:

palabras += NF

en términos generales, si se tiene una expresión de la forma:

var = var op exp

donde:

var = nombre de variable
op = algún operador aritmético
exp = expresión

se puede transformar a:

var op= exp

por lo tanto se tienen los operadores +=, -=, *=, %= y /=.

Los patrones en awk, pueden involucrar operadores aritméticos; pero también pueden ser expresiones que incluyan otro tipo de operadores, como lógicos, relacionales o de evaluación de expresiones regulares. Por ejemplo, para especificar un patrón que seleccione los registros de entrada que tienen cinco campos, se utilizaría la siguiente expresión:

NF == 5

Un patrón que seleccione registros cuya longitud máxima sea de 80 caracteres:

```
length($0) <= 80
```

Para seleccionar registros que tengan solamente 10 caracteres se podrían utilizar alguno de los siguientes patrones:

```
$0 ~ /^.........$/
length($0) == 10
```

Un patrón que seleccione los registros pares con cinco campos cuya longitud total no sea mayor a 80 caracteres sería:

```
NR % 2 == 0 && NF == 5 && length($0) <= 80
```

En el ejemplo anterior, podría surgir la pregunta ¿que operadores se evalúan primero?, para contestarla hay que saber que todos los operadores tienen cierta precedencia que indica el orden de evaluación. Si se quiere romper dicha precedencia, se pueden utilizar paréntesis. La tabla 5.4 muestra los operadores válidos en awk en orden creciente de precedencia.

OPERADOR	DESCRIPCION
= += -= *= %= /=	Asignación
||	OR lógico. Se aplica la regla del corto circuito.
&&	AND lógico. Se aplica la regla del corto circuito.
!	NOT lógico. Negación.
> < >= <= == != ~ !~	Operadores relacionales. Los dos últimos se aplican en expresiones regulares, para denotar correspondencia y no correspondencia.
+ -	Suma y resta.
* / &	Multiplicación, división y residuo.
++ --	Incremento y decremento unitario.

Tabla 5.4. Operadores en awk por orden creciente de precedencia.

2. Control de flujo

Existen algunas proposiciones en awk para especificar un orden en la realización de las operaciones de un programa, dichas proposiciones son las estructuras de control de flujo de la programación estructurada.

La primera de ellas es la proposición if-else, la cual tiene la siguiente sintaxis:

```
if (expresión)
     proposición1
else
     proposición2
```

donde las proposiciones 1 y 2 son expresiones o un bloque que consiste en expresiones encerradas entre llaves y la parte else es opcional. La expresión se evalúa, si es verdadera (si la expresión tiene un valor diferente de cero), se ejecuta la proposición1; si es falsa (el valor de la expresión es cero) y si existe la parte else, se ejecuta la proposición2.

El programa prog1.awk mostrado en la figura 5.2 obtiene el archivo más grande así como el más pequeño al pasarle como flujo de datos de entrada un listado con información de los archivos de un directorio. La forma de invocarlo, para el directorio de trabajo, es la siguiente:

```
% ls -la | awk -f prog1.awk
```

```
{       if ( NR == 1 )
                next
        if ( NR == 2 ) {
                min = $4
                max = $4
                archMax = $8
                archMin = $8
        }
        else    {
                if ( $4 > max ) {
                        max = $4
                        archMax = $8
                }
                if ( $4 < min ) {
                        min = $4
                        archMin = $8
                }
        }
}
END {
        printf "Archivo mas grande: %s %7.2f kbytes\n",
archMax, max/1000
        printf   "Archivo   mas   pequenio:   %s   %7.2f
kbytes\n",                  archMin,min/1000
}
```

Fig. 5.2. Listado del programa prog1.awk

El comando ls genera, como ya se habia visto, un listado con información para los archivos; sin embargo, la primera línea de salida es un encabezado que indica el total de archivos listados, por lo cual esta primera línea no debe procesarse. La instrucción **next** provoca que se lea el siguiente registro en el flujo de datos de entrada.

Otro programa ejemplo se muestra en la figura 5.3 y es una implementación con awk del comando wc.

El comando awk, también incluye proposiciones que permiten realizar ciclos, dichas proposiciones son **while** y **for**. La sintaxis para cada una de ellas se muestra a continuación:

```
while (expresión)
    proposición
```

```
BEGIN {
        archivo = FILENAME
}
{
      if ( archivo != FILENAME )
      {
         printf"%8d%8d%8d  %s\n",
               NR-1,palabras,caracteres,archivo
         palabras = 0
         caracteres = 0
         NR = 1
      }
      caracteres += length($0) + 1
      palabras += NF
      archivo = FILENAME
         totalCar += length($0) + 1
         totalPal += NF
         totalLin++
}
END {
      printf"%8d%8d%8d %s\n",
            NR,palabras,caracteres,FILENAME
      printf"%8d%8d%8d
total\n",totalLin,totalPal,totalCar
}
```

Fig. 3.3. Implementación de wc con awk.

```
for(expresión1;expresión2;expresión3)
        proposición
```

Para el caso de while, se evalúa la expresión si esta es verdadera se ejecutan las instrucciones que forman la proposición y se vuelve a evaluar la expresión. El ciclo continua hasta que la expresión sea falsa, momento en el cual la ejecución del programa

continua después de la proposición. Con la proposición while se puede implementar el comportamiento del for de la siguiente forma:

```
expresión1
while (expresión2) {
     proposición
     expresión3
}
```

La expresión1 en el for generalmente es una inicialización y solamente se realiza una vez, al comienzo del for; después se evalúa la expresión2 si esta es verdadera se ejecutan las instrucciones que forman la proposición y finalmente se realiza la expresión3, para posteriormente volver a evaluar la expresión2. El ciclo continua hasta que la expresión2 sea falsa.

3. Arreglos

El lenguaje de programación de awk permite manejar arreglos. Al igual que las variables, los arreglos no se necesitan inicializar y pueden ser arreglos numéricos o de cadenas de caracteres. La inicialización funciona igual que con las variables.
La figura 5.4 muestra el programa sort.awk que ordena las líneas de entrada por el método conocido como "la burbuja"; en dicho ejemplo se almacenan las líneas de entrada en un arreglo de cadenas de caracteres.

```
# Programa de ordenamiento de las líneas de entrada

{       linea[NR] = $0 }

END {
        for(i=1;i<NR;i++)
                for(j=NR;i<j;j--)
                        if (linea[j-1] > linea[j]) {
                                aux = linea[j]
                                linea[j] = linea[j-1]
                                linea[j-1] = aux
                        }
        for(i=1;i<=NR;i++)
                print linea[i]
}
```

Fig. 5.4. Programa sort.awk

Normalmente, los índices de los arreglos son valores enteros;
sin embargo, el lenguaje de programación de awk permite manejar como índice cualquier valor. Cuando los índices de los arreglos no son valores enteros se habla de los llamados arreglos asociativos. Cuando se manejan arreglos asociativos, podría surgir la pregunta ¿como se recorre el arreglo, si los índices no llevan un orden?. Los índices para los arreglos asociativos tienen un orden imprevisto y awk utiliza un esquema de hashing para garantizar que el acceso a cualquier elemento del arreglo tarde mas o menos el mismo tiempo. Para recorrer todos los elementos del arreglo se utiliza una variante de la proposición for, cuya sintaxis es la siguiente:

for(índice in arreglo)
proposición

en esta proposición for cada uno de los valores de los índices se van asignando en cada iteración a la variable índice, hasta que el arreglo denominado arreglo se recorre completamente. El orden en que se van obteniendo los índices es impredecible.

Un ejemplo que podría aclarar el uso de los arreglos asociativos, lo constituye el programa que obtiene la frecuencia que tiene cada una de las palabras de un texto. En principio no se sabe cuantas palabras contiene un texto cualquiera, si se contara únicamente con arreglos con índices enteros se tendría que tener un mapeo de un índice a una palabra; sin embargo, con

los arreglos asociativos esto es innecesario ya que las mismas

```
# Programa que obtiene la frecuencia de las palabras
# de una entrada de datos

{     for(i = 1;i <= NF;i++)
            palabras[$i]++
}
END {
      for(i in palabras)
            printf("%s %d\n",i,palabras[i]
}
```

Fig. 5.5. Programa que obtiene la frecuencia de las palabras del flujo de datos de entrada.

palabras son los índices y los elementos de los arreglos contienen la frecuencia. De esta forma el programa que realiza tal función se muestra en la figura 5.5.

4. La nueva versión de AWK

Las implementaciones más recientes de UNIX, incluyen una nueva versión de AWK, comúnmente llamada "nuevo AWK". Esta nueva versión, es compatible con la descrita en este capítulo. En algunos sistemas, se cuenta con las dos versiones: **nawk** y **oawk**, que

representan a la nueva versión y a la descrita en este capítulo

respectivamente. El comando awk es una liga a nawk o oawk.

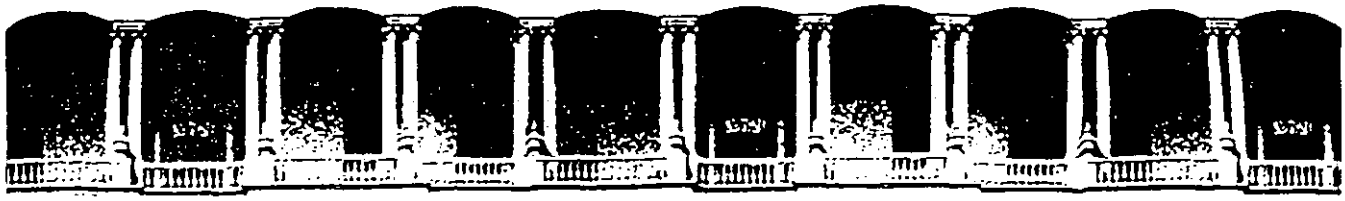
Algunas extensiones de la nueva versión de AWK se mencionan a continuación:

- Se pueden definir funciones.

- Se pueden hacer interconexiones de comandos en el programa.

- Se pueden borrar a tiempo de ejecución arreglos asociativos.

- El parser para el nuevo AWK elimina algunas ambiguedades que se permiten en la versión anterior.



FACULTAD DE INGENIERIA U.N.A.M.
DIVISION DE EDUCACION CONTINUA

SISTEMA OPERATIVO UNIX

PARTE III

MATERIAL DIDACTICO

SEPTIEMBRE 1994

Palacio de Minería Calle de Tacuba 5 Primer piso Deleg. Cuauhtémoc 06000 México, D.F. Tél.: 521-40-20 Apdo. Postal M-2285

El uso del shell

Los componentes o la arquitectura de alto nivel de UNIX se pueden representar por el diagrama de la figura 2. Si vemos el sistema como un conjunto de capas, el sistema operativo propiamente dicho conforma la capa más interna, esta parte del sistema operativo es la que se comunica directamente con el hardware y se le conoce comúnmente como **kernel** o **núcleo**.' Entre las funciones del núcleo se pueden mencionar las siguientes: planificación de tareas, administración de recursos del sistema, administración de procesos y manejo de memoria.

En la siguiente capa se encuentran las utilerías y programas de aplicación vinculados con el sistema que se encargan de ejecutar una variedad de rutinas y funciones especiales de mantenimiento del sistema. Estas utilerías se comunican con el kernel por medio de una interfase desarrollada en lenguaje C que se conoce como **llamadas al sistema**. Muchas de las utilerías y programas forman parte de la configuración estándar de UNIX y son conocidos comúnmente como comandos.

Uno de los programas de aplicación más importantes es el intérprete de comandos o **shell**. Este programa se ejecuta inmediatamente después de que se abre una sesión de UNIX, es un proceso que interpreta los comandos que teclea el usuario y ejecuta las acciones asociadas con dichos comandos; de esta forma se lleva a cabo la comunicación entre el usuario y el sistema operativo.

Existen varios tipos de shell entre los que se encuentran *Bourne shell*, *C-shell*, *Reduced shell*, *Korn shell* y *Visual shell*. El usuario puede desarrollar su propio interprete de comandos para que sea este el que lo comunique con el núcleo.

En la capa superior de la arquitectura de UNIX se encuentran los programas de aplicación que no están comprendidos dentro de la configuración estándar de UNIX, es decir programas creados por los usuarios.

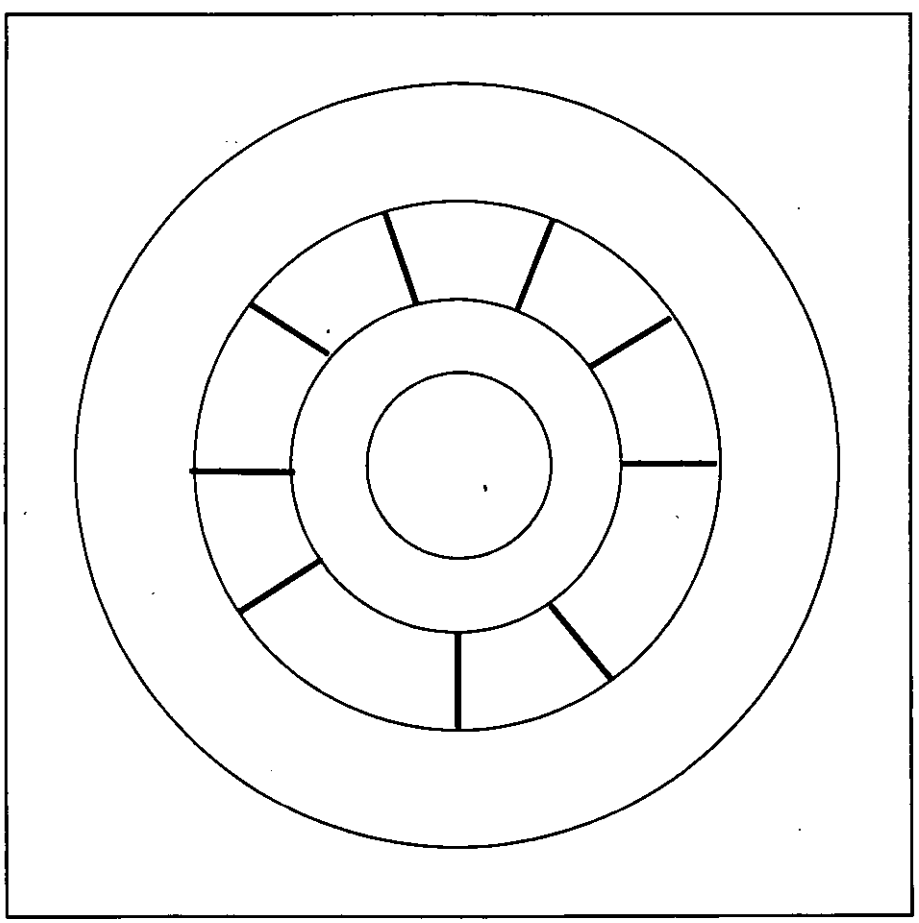


Fig. 2. Arquitectura de UNIX

Cuando se inicia una sesión Unix, el sistema operativo proporciona un shell por default, este shell se denomina login shell. El login shell se indica en el archivo /etc/passwd y no necesariamente debe de ser un shell, podría ser cualquier programa ejecutable.

Mientras uno se encuentra en sesión se puede mandar a ejecutar algún otro shell para realizar algunos trabajos, pero estos procesos serán procesos '*hijos*' de nuestro login shell.

Si se desea podemos cambiar de login shell, ésto se hace mediante el comando *chsh*, al cual se le proporciona la cuenta a la que se le desea cambiar de login shell y la ruta completa de donde se encuentra localizado el intérprete de comandos (shell). Por ejemplo, el usuario raq desea cambiar su login shell a C shell (csh), entonces debe teclear lo siguiente:

$ chsh acf /bin/csh

Esto funciona para las versiones de Unix system V. En Ultrix basta con dar el comando chsh y presionar return. La computadora nos informará cual es nuestro login shell actual y nos preguntará cual deseamos que sea nuestro nuevo login shell, donde basta con teclear la abreviación del shell que deseamos. Por ejemplo:

$ chsh
Changing login shell for acf
shell [/usr/bin/ksh]: **csh**

Una vez echo ésto, la próxima vez que entremos a sesión, nuestro login shell será C shell (csh).

Cada vez que un usuario entra a sesión, la computadora ejecuta algunos cuantos archivos de comandos para definir ciertas características del medio ambiente donde trabajará el usuario, es decir del shell..

Los archivos de configuración dependen del login shell, pero pueden ser utilizados para establecer el valor de variables de ambiente como podría ser el path o algunos alias o variables que se quisieran definir de manera global. Cuando se creen o modifiquen los archivos de configuración, debemos cerciorarnos de que tengan permiso de ejecución.

Agrupando comandos del shell

Hasta ahora, hemos discutido la forma de mandar a ejecutar varios comandos y/o archivos de comandos, pero de momento solo se ha mandado a ejecutar un comando a la vez. Uno puede obtener más ventajas sobre el shell desarrollando la habilidad de ejecutar varios comandos a la vez.

Siempre que usamos el shell con la entrada estándar, el teclado, y con la salida estándar, el monitor, estamos usando el shell en forma interactiva. Una de las ventajas de usar el shell en forma interactiva, es la de poder observar el resultado de los comandos en cuanto nuestra terminal los recibe. Una de las desventajas de usar el shell en forma interactiva (hasta ahora) es la de poder mandar ejecutar un sólo comando a la vez.

Si mandamos ejecutar un segundo comando antes de que el primero termine de ejecutarse, la salida del primero se mezclará muy probablemente en la terminal con nuestro segundo comando mientras lo vamos escribiendo. En el siguiente ejemplo, al escribir el comando **date** se mezcla la salida del comando **who**:

```
% who
sjg000          tty00   Apr 26 15:03
pnh000          tty02   Apr t26 14:55
kjd000          tty04   Apr 26 12:44
%
```

Muchas veces necesitamos observar el resultado de un comando para decidir cuál será el siguiente comando a ejecutar. Sin embargo, hay ocasiones en que la salida del comando anterior no tiene efecto sobre el siguiente comando a teclear. Habrá ocasiones en las que deseemos ejecutar varios comandos a la vez ya que no importa el resultado de éstos para que los demás se ejecuten. Esto es, es posible mandar a ejecutar varios comandos a la vez.

Hasta ahora el 'separador' de comandos ha sido la tecla RETURN, o ENTER. Debido a que RETURN es la tecla que le indica al shell empezar la ejecución del comando escrito en la línea de comandos, sólo se puede ejecutar un comando a la vez.

Es posible mandar a ejecutar varios comandos a la vez separando los comandos por puntos y comas (;) como se muestra más adelante. Cuando se le pasan al shell varios comandos para su ejecución separados por puntos y comas en la misma línea de comandos, ésto se conoce como un 'job'. Más adelante en este capítulo hablaremos sobre el manejo de jobs.

```
% who;date;echo "hola"
sjg000          tty00  Apr 26 15:03
pnh000          tty02  Apr 26 14:55
kjd000          tty04  Apr 26 12:44
Mon Apr 26 15:15:32 GMT 1993
hola
```

Además, podemos continuar escribiendo un comando en la siguiente línea, utilizando diagonales invertidas (\) al final de cada línea seguida de la tecla RETURN. Con ésto se consigue 'escapar' el significado de la tecla RETURN ante el shell para que éste no mande a ejecutar los comandos escritos hasta el momento, y poder seguir escribiendo en la siguiente línea. Esto es especialmente útil cuando se trata de comandos largos como por ejemplo:

```
% grep "estados" capítulo1.txt capítulo2.txt \<RETURN>
capítulo3.txt ../pubicaciones.old contenido.list
```

Combinaciones de puntos y comas y antidiagonales pueden ser usadas para escribir comandos que no estén limitados por la longitud de la línea de comandos como se muestra a continuación:

```
% echo "La fecha y hora actual es: "; date; \<RETURN>
echo "Lista de los usuarios en sesion:";\<RETURN>
who; echo "Unix posee varias caracteristicas interesantes."
La fecha y hora actual es:
Mon Apr 26 17:11:02 GMT 193
Lista de los usuarios en sesion:
sjg000          tty00  Apr 26 15:03
pnh000          tty02  Apr 26 14:55
kjd000          tty04  Apr 26 12:44
Unix posee varias caracteristicas interesantes.
%
```

Es importante notar las diferencias entre usar punto y coma (;) para separar comandos y las interconexiones (|) o pipes. Las interconexiones le dicen al shell que la salida del primer comando la use como la entrada del segundo comando. Un punto y coma entre dos comandos le dice al shell que ejecute el primer comando y después de éste ejecute al segundo comando. Esto es, mientras que con interconexiones los procesos se realizan en forma concurrente, con puntos y comas se ejecutan de manera secuencial.

Cuando se usan interconexiones (|) entre comandos en lugar de puntos y comas, solo es desplegada la salida del último comando. Esto es debido a que la interconexión toma la salida del comando que se encuentra antes de la interconexión y esta se convierte en la entrada del comando que se encuentra después de la interconexión. Si el comando que se encuentra después de la interconexión generalmente no toma su entrada del teclado, la interconexión no tiene sentido. Por ejemplo:

```
% date | who | pwd | ls
ascii   calendar        mbox        plum        settings
bin         misc            practice    shell
% date ; who ; pwd; ls
Mon Apr 26 17:15:00 GMT 1993
sjg000          tty00  Apr 26 15:03
kjd000          tty04  Apr 26 12:44
/users/jess
ascii   calendar        mbox        plum        settings
bin         misc            practice    shell
%
```

Si por el contrario, usamos puntos y comas en lugar de utilizar interconexiones, los resultados pueden ser confusos. Por ejemplo, los siguientes comandos **date** y **who** son ejecutados, pero el resultado del comando who no es ordenado como se esperaba. En lugar de ésto, da la impresión de que la terminal se 'bloqueo' (queda esperando indefinidamente), lo cual se debe a que el comando sort está esperando datos de la entrada estándar (el teclado). Debemos, entonces, presionar la tecla de interrupción o la de fin de archivo para recibir nuevamente el prompt. Si se usan interconexiones, la salida del comando who entonces es ordenada como se esperaba y el prompt es regresado inmediatamente después.

```
% date ; who ; sort
Mon Apr 26 17:15:00 GMT 1993
sjg000              tty00   Apr 26 15:03
pnh000              tty02   Apr 26 14:55
kjd000              tty04   Apr 26 12:44
^D
%
% date ; who | sort
Mon Apr 26 17:15:43 GMT 1993
kjd000              tty04   Apr 26 12:44
pnh000              tty02   Apr 26 14:55
sjg000              tty00   Apr 26 15:03
%
```

También podemos agrupar comandos entre paréntesis. El efecto que esto tiene es el de generar un nuevo shell ('hijo' o sub-shell) a nuestro shell actual, y mandar a ejecutar los comandos tecleados en este nuevo sub-shell. Por ejemplo, supongamos que nos encontramos trabajando en el subdirectorio /users/alumnos/vlu000 viendo algunos programas fuente, y deseamos examinar el contenido de algunos archivos en el subdirectorio /users/pub. Entonces podríamos ejecutar el comando siguiente:

```
% (cd /users/pub; grep 'printf' snmpd.c ifconfig.c)
```

Cuando el prompt aparezca de nuevo, el directorio de trabajo permanecerá igual que antes de haberse ejecutado el comando anterior. Esto es debido a que el comando cd y el comando grep fueron ejecutados en el nuevo sub-shell. En cuanto éstos terminaron, el nuevo sub-shell también terminó y el control fue regresado al shell que mandó a ejecutar el comando. En ese shell (el actual) no se ejecutó nunca ninguna orden de cambiar de directorio, es por eso que al regresar el control, se sigue trabajando en el mismo directorio.

También podemos utilizar el agrupamiento entre paréntesis para realizar direccionamientos de salida estándar y de error a la vez. Cosa que no es posible realizar utilizando en forma independiente los símbolos de redireccionamiento. Por ejemplo, para redireccionar la salida estándar de un comando grep al archivo *salida* y la salida de errores al archivo *diagfile* se puede utilizar el comando siguiente:

% **(grep 'mount' /etc/* > salida) >& diagfile**

Cabe notar lo siguiente:

1.- La salida estándar del comando grep se mando al archivo salida:

2.- La salida de errores (**>&**) del comando encerrado entre paréntesis se mando al archivo diagfile:

Shell scripts

Los shell scripts son también conocidos como programas de shell (shell programs) o procedimientos de shell (shell procedures). Para la creación de shell scripts todo lo que debemos hacer es crear un archivo que contenga comandos de Unix ordinarios en el orden en que normalmente se proporcionarían.

Los comandos que se escriben dentro de un shell script son los mismos que se usan desde la línea de comandos, sin embargo, debido a que existen varios intérpretes de comandos como son el C-shell (csh), el Bourne-shell (sh), el Korn-shell (ksh) y otros más, ciertas instrucciones, relacionadas principalmente con el control de flujo, deberán de ser escritas con la sintaxis propia del intérprete de comandos encargado de ejecutar nuestro archivo de comandos.

Por ejemplo, si quisiéramos que la computadora nos diera un listado del número y nombres de los subdirectorios que se encuentran en nuestro home directory, podríamos utilizar comandos de Unix como los siguientes:

```
% cd
% ls -l | grep '^d' | wc -l          Contar el número de subdirectorios
```

```
...
% ls -l | grep '^d'                    Listar solo los subdirectorios
...
%
```

Pero esto se puede hacer con un solo comando si nosotros colocamos todos estos comandos dentro de un shell script. El siguiente shell script es usado para contar y listar los subdirectorios en nuestro home directory:

```
# /bin/csh
# 'Este comando cuenta el numero de subdirectorios en el'
# 'home directory de un usuario, imprime el numero, y lista'
# 'los nombres de todos los subdirectorios.'
#
cd
echo -n "Tu home directory es "
pwd
echo -n "El numero de subdirectorios en tu home directory es "
ls -l | grep '^d' | wc -l;   # 'cuenta numero de directorios'
echo "Estos son:"
ls -l | grep '^d';          # 'lista los directorios'
```

Algunos puntos a destacar sobre este archivo son:

- Es importante documentar los programas propiamente. Es una muy buena costumbre colocar al inicio del archivo la ruta completa del intérprete de comandos que ejecutará nuestro shell script. En este caso se trata del intérprete C-shell (/bin/csh). Esto es muy útil, sobre todo cuando después de un tiempo editamos nuestro archivo, esto nos recordará para que intérprete de comandos está escrito evitándonos grandes dolores de cabeza por utilizar la sintaxis de otro intérprete de comandos.

- El símbolo de número (#) es usado para introducir comentarios. Los comentarios no son interpretados por el shell como comandos. Son ignorados.

- Cada línea de comentario es encerrada entre apóstrofos. Así se asegura que el comentario sea efectivamente tratado como comentario.

- Para que el shell script sea ejecutado por el intérprete de comandos C-shell, es muy importante que el primer carácter del primer renglón contenga el símbolo de número (#). Si no es así el intérprete que ejecutaría nuestro archivo de comandos sería el Bourne-shell.

- Un comentario que está en la misma línea de un comando es llamado un comentario en línea (in-line comment). Hay que asegurarse de separar el comando del comentario con un separador de comandos punto y coma (;). Si no se usa, el comentario es interpretado como un argumento del comando.

- La opción -n del comando echo le dice al shell no imprimir un retorno de carro al final de la línea impresa.

Ejecutando Shell Scripts.

Hay tres formas de ejecutar un shell script:

1.- Invocar un subshell.
2.- Correr el script en el shell actual.
3.- Hacer el script ejecutable.

Invocando un subshell.

Para ejecutar un shell script en un subshell, escribimos el nombre del shell que ejecutará nuestro shell script pasándole como parámetro el nombre del script, en este caso para C-shell:

```
% csh nombre_archivo
```

donde nombre_archivo se refiere al nombre del archivo del shell script. Para el caso del script anterior (cld) lo pudimos haber mandado ejecutar con la siguiente instrucción:

```
% csh cld
```

Corriendo el script en el shell Actual.

Este método tiene el mismo efecto que ejecutar todos los comandos del shell separados por punto y comas y diagonales invertidas. Para hacer esto utilizamos el comando *source* de la siguiente manera:

```
% source nombre_archivo
```

Para el caso del script anterior (cld) lo pudimos haber mandado ejecutar con la siguiente instrucción:

```
% source cld
```

Haciendo el Script Ejecutable.

El tercer método de ejecutar un shell script es hacer el script ejecutable usando el comando chmod (agregar el permiso de ejecución), es decir para convertir los shell scripts a comandos, todo lo que se tiene que hacer es hacerlos ejecutables con este comando. Por ejemplo para convertir el shell script cld, en un comando que se pueda ejecutar, se debe realizar lo siguiente:

```
% chmod ugo+x cld

% cld
...
%
```

Esta forma del comando chmod le dice al shell sumar (**+**) permiso de ejecución (**x**) al usuario (**u**), grupo (**g**), y otros (**o**).

El hacer un shell script ejecutable no tiene efecto sobre la habilidad para usar *csh*, y *source* para ejecutarlo. Todos estos comandos se comportarán exactamente como lo hicieron antes de que el archivo fuera ejecutable.

Por otra parte hay que señalar que, los comandos sólo pueden ser ejecutados si la variable path contiene a el directorio actual de trabajo (representado por un punto '.') o si se proporciona el pathname completo.

Programación con C-Shell

Usando Variables dentro de los shell script.

Una variable permite "llenar el espacio en blanco" en tiempo de ejecución. Es posible usar variables para almacenar información interactivamente además de ser usadas para almacenar información dentro de los shell scripts. Hay dos formas para que una variable obtenga un valor:

- A través de un sentencia de asignación que es escrita dentro del shell script.
- A través de un sentencia de lectura en la cual el usuario proporciona el valor cuando el shell script es ejecutado.

Las sentencias de asignación las podemos escribir dentro de los shell scripts utilizando la sentencia *set* en el siguiente formato:

```
% set <nombre variable>=<valor>
```

Una asignación de este tipo busca un lugar de almacenamiento con la etiqueta <nombre variable> y almacena ahí el valor <valor> especificado.

Los nombres de variables deben empezar con una letra y contener sólo letras, números, y subguiones. Pueden ser de hasta 20 caracteres de largo. Por convención, a las variables se les dan nombres mnemónicos. Mnemónico significa que el nombre de la variable te dice algo acerca de lo que la variable contiene.

Consideremos un shell script que usa el comando grep para buscar ocurrencias de palabras mal escritas. En tal script, podemos encontrar el siguiente comando:

```
set palabra="teh"
```

el cual almacena el valor "teh" en una variable llamada palabra. Si después en el shell script, aparece el comando:

```
set palabra="tej"
```

el contenido de la variable palabra será reemplazado por el valor "tej". Las comillas alrededor de "teh" y "tej" indican que el valor de adentro debe ser tomado literalmente, incluyendo cualquier carácter de espacio en blanco. Una cadena de caracteres encerrada entre comillas es conocida como una cadena literal.

Suponiendo que quisiéramos borrar una variable de su lugar de almacenamiento podemos usar el comando *unset*:

```
unset <nombre variable>
```

Además de almacenar información como el valor de una variable, se debe poder recuperar esa información. Para referirnos al valor de una variable, debemos poner el signo ($) inmediatamente antes del nombre de la variable (por ejemplo, $palabra). Para imprimir el

valor de una variable escribimos:

echo $<nombre variable>

El siguiente ejemplo asigna a la variable 'mensaje' la cadena de caracteres "Seleccione una opción", después, visualiza el contenido de la variable 'mensaje' y después desasigna esta variable eliminándola. Por último verifica que la variable ya no existe.

```
% set mensaje="Seleccione una opción."
% echo $mensaje
Seleccione una opción.
% unset mensaje
% echo $mensaje
%
```

Si omitimos el signo $, sólo el nombre de la variable será impreso:

```
% echo mensaje
mensaje
```

Se puede imprimir a la vez el valor de más de una variable en un enunciado echo. Se puede inclusive combinar variables y texto:

```
% set primero=1
% set segundo=2
% echo "El valor de la primera variable es $primero"
El valor de la primera variable es 1
%
% echo "Los valores de la primera y segunda variables";\
echo "son $primero y $segundo respectivamente."
Los valores de la primera y segunda variables
son 1 y 2 respectivamente.
```

Hay que notar que en el ejemplo anterior, las comillas permiten que los valores de variables sean substituidos dentro de cadenas literales. Los apóstrofes no permiten esta substitución.

El <valor> en una sentencia de asignación no tiene que ser una cadena literal. Puede ser también el valor de otra variable, la salida de un comando, o cualquier combinación de estos elementos. Por ejemplo el siguiente comando:

```
% set prompt="`pwd` $user > "
```

asigna a la variable prompt el resultado de la ejecución del comando *pwd* (print work directory), así como el valor de la variable *$user* además de un símbolo 'mayor que'.

Otra forma de asignar un valor a una variable es tener al usuario proporcionando el valor. Este tipo de asignación es conocido como una sentencia read (read statement). La sintaxis de esta sentencia es la siguiente:

```
% set <nombre variable>=$<
```

Cuando el shell encuentra una sentencia read espera hasta que escribamos el final de una línea con un RETURN, antes de continuar interpretando el shell script.

El siguiente es un ejemplo de la implementación del comando ADD el cual nos permite agregar información a una agenda telefónica:

```
#!/bin/csh
# 'Archivo: ADD -- Agrega información a la mini agenda telefónica'
#
# 'Este comando es usado para agregar información a la'
# 'agenda telefónica contenida en el archivo: datafile. Cada'
# 'registro consiste de un nombre y apellido, seguido'
# 'de una fecha de nacimiento y un numero telefónico.'
# 'El usuario que corra este archivo de comandos le será'
# 'preguntada toda la información necesaria'
#
echo -n "Cual es el nombre de la persona a agregar? "
set nombre=$<
echo -n "Cual es el apellido de $nombre? "
set apellido=$<
echo "Teclea la fecha de nacimiento en el formato mes/día/año."
set dian=$<
echo -n "Cual es el teléfono de $nombre $apellido: "
set telefono=$<
echo "$nombre $apellido<TAB><TAB>$dian<TAB><TAB>$telefono"\
>> $HOME/datafile
sort +1 -2 $HOME/datafile > $HOME/temp
mv $HOME/temp $HOME/datafile
echo "El siguiente registro ha sido agregado a tu base de datos:"
echo "$nombre $apellido<TAB><TAB>$dian<TAB><TAB>$telefono"
```

NOTA: Dentro algunas sentencias echo en el ejemplo anterior y en algunos posteriores, aparece la secuencia de caracteres '**<TAB>**', estos deberán de substituirse por un tabulador en lugar de escribir literalmente esta cadena de caracteres.

Diferencias entre ' , " y `.

Estas marcas preservan los espacios en blanco en un comando echo y en expresiones regulares para grep. También previenen que el shell interprete metacaracteres de nombres de archivos.

Adicionalmente a los apóstrofos y a las comillas, hay un tercer tipo de marca que debemos conocer y que es interpretado de manera diferente que los otros dos. El tercer tipo es un apóstrofo al revés o acento grave (`). Es usado para forzar que el shell interprete un comando como en los siguientes dos ejemplos:

```
% echo "La fecha y hora actual es `date`."
La fecha y hora actual es Mon May 14 14:55:39 EDT 1993.
%
% set hoy=`date`
% echo "La fecha y hora actual es $hoy."
La fecha y hora actual es Mon May 14 14:55:39 EDT 1993.
```

Los efectos de los tres tipos de marcas se explican a continuación:

- Los apóstrofos (') protegen todo. Cualquier cosa encerrada en apóstrofos es tomada literalmente.

- Las comillas (") protegen espacios en blanco, metacaracteres de nombres de archivos, y apóstrofes. No protegen variables (precedidas por $) o comandos encerrados por apóstrofes al revés (`). El signo de pesos y los apóstrofes al revés son siempre interpretados.

- Los acentos graves le dicen al shell que ejecute el comando encerrado adentro. Todo lo que se encuentra adentro de ellos es interpretado tal y como si fuera tecleado interactivamente.

Usando Archivos Temporales.

Cuando se usan archivos temporales hay que tener cuidado de que no exista un archivo con el mismo nombre que el del temporal en el directorio de trabajo actual. Si existe, el shell script lo borrará. Además, si a otro usuario se le ocurre crear un archivo temporal, muy probablemente use también el nombre temp, con lo cual podría borrar nuestra información si es que se usa el mismo subdirectorio para escribir este tipo de archivos temporales. Obviamente esto no es lo que deseamos.

Afortunadamente hay alternativas para crear un archivo temporal en el directorio actual o en algún otro subdirectorio. El directorio de sistema /tmp existe solamente con el propósito de proporcionar espacio para archivos temporales, ya sean estos de comandos, de información o de lo que sea.

Hay que asegurarnos que cuando algún comando cree un archivo temporal, el nombre que le demos a ese archivo sea único. Se puede crear un nombre de archivo único agregándole el PID (Process IDentification) del comando al nombre del archivo. La variable de shell $$ contiene el valor del número de identificación del proceso en ejecución. Si le damos a un archivo creado dentro de un shell script el nombre $0$$, el nombre del comando (el nombre del archivo que contiene al script) es substituido por $0, y el PID es substituido por $$. Si se recibe un mensaje de error que diga "Variable syntax", deberemos llamar al archivo ${0}$$.

El siguiente ejemplo es una versión modificada del archivo de comandos ADD, donde hacemos uso de lo que se acaba de señalar. Esta nueva versión de ADD también verifica si la persona a dar de alta ya existe dentro del archivo $HOME/datafile para evitar duplicar información:

```
#!/bin/csh
# 'Archivo: ADD -- Agrega información a la mini agenda telefónica'
#
# 'Este comando es usado para agregar información a la'
# 'mini agenda telefónica contenida en el archivo: datafile. Cada'
# 'registro consiste de un nombre y apellido, seguido'
# 'de una fecha de nacimiento y un numero telefónico.'
# 'El usuario que corra este archivo de comandos le será'
# 'preguntada toda la información necesaria'
#
clear
echo -n "Cual es el nombre de la persona a agregar? "
set nombre=$<
echo -n "Cual es el apellido de $nombre ? "
set apellido=$<
if { grep -sq "$nombre $apellido" $HOME/datafile } then
  echo "$nombre $apellido ya se encuentra en su agenda."
else
  echo "Teclea la fecha de nacimiento en el formato mes/día/año."
  set dian=$<
  echo -n "Cual es el teléfono de $nombre $apellido: "
  set telefono=$<
  echo "$nombre $apellido<TAB><TAB>$dian<TAB><TAB>$telefono" \
    >> $HOME/datafile
  sort +1 -2 $HOME/datafile > /tmp/${0}$$
  mv /tmp/${0}$$ $HOME/datafile
  echo "El siguiente registro ha sido agregado a tu base de \
datos:"
  echo "$nombre $apellido<TAB><TAB>$dian<TAB><TAB>$telefono"
endif
```

Observemos que ahora el archivo temporal se mando al subdirectorio /tmp bajo un nombre que será único (${0}$$).

Control de Flujo.

Alguna vez , quizás deseemos que la computadora ejecute algunos comandos sólo bajo condiciones particulares o para repetir ciertos comandos. Para tener a la computadora ejecutando comandos en orden no-secuencial o no-lineal, se necesita alterar el flujo de control del shell script. Las construcciones de control de flujo son comandos especiales que se pueden usar en los shell scripts (o en cualquier programa de computadora) para hacer que los comandos sean ejecutados condicionalmente o repetidamente.

Uno de los usos más importantes de la ejecución condicional en programas de computadora es el checar errores. Usando sentencias condicionales, los programas pueden checar una entrada de usuario para detectar diferentes tipos de errores y mandar un mensaje de aviso adecuado al usuario.

Los símbolos && y || permiten ejecutar un segundo comando si el comando previo tiene éxito o falla respectivamente. Por ejemplo:

```
% echo "hola" && echo "adiós"
hola
adiós

% echo "hola" || echo "adiós"
hola
%
```

En el primer comando, le decimos al intérprete de comandos que ejecute el segundo comando (echo "adiós"), solo si el primer comando (echo "hola") tiene éxito; por supuesto esta primer sentencia echo siempre tiene éxito y la segunda sentencia echo es ejecutada. En el segundo caso (||) le decimos que ejecute el segundo comando (echo "adiós") solo si el primer comando (echo "hola") falla. Como el primer comando echo no falla, el segundo comando echo no es ejecutado.

Pero muchas veces es necesario poder mandar ejecutar un *grupo* de instrucciones en caso de éxito o falla de algún comando, no solo un solo comando como en el caso anterior.

Consideremos el script de comandos DELETE que se muestra en el siguiente ejemplo, el cual tiene la finalidad de borrar un registro de nuestra agenda telefónica. Idealmente, se quiere borrar el nombre sólo si está en la base de datos e imprimir un mensaje si no está. Esta revisión se realiza mediante la estructura if-then-else, en caso de tener éxito el comando que se encuentra entre llaves {}, se ejecutarán las instrucciones grep, mv y echo; en caso de falla, se ejecutará solamente el comando echo que se encuentra después del else.

```
#!/bin/csh
# 'Archivo: DELETE -- BORRA un registro de la agenda telefónica.'
#
echo -n "Cual es el nombre de la persona a borrar? "
set nombre=$<
echo -n "Cual es el apellido de la persona a borrar? "
set apellido=$<
if { grep -sq "$nombre $apellido" $HOME/datafile } then
  grep -v "$nombre $apellido" $HOME/datafile > temp
  mv temp $HOME/datafile
  echo "$nombre $apellido ha sido borrado(a) de tu base de datos."
else
  echo "$nombre $apellido no se encuentra en su agenda."
endif
```

La estructura de control if/then permite checar que SI (IF) el primer comando tiene éxito ENTONCES (THEN) ejecute los siguientes
comandos. El final de la lista de comandos es marcado con un terminador. El terminador es endif en el caso del C shell. La sintaxis es la siguiente:

```
if {<comando>} then
  <lista de comandos>
endif
```

NOTA: La colocación de la palabra then es muy importante. En C shell then debe ser parte de la misma línea del if.

La sintaxis de la construcción if/then/else es:

```
if {<comando>} then
  <lista de comandos>
else
  <lista de comandos>
endif
```

Las estructuras de control if/then e if/then/else nos permite escribir shell scripts que efectúan otros tipos de decisión. El formato más general para una sentencia condicional es:

```
if {<condición>} then
  <lista de comandos>
else
  <lista de comandos>
endif
```

La <condición> puede ser de varios tipos:

- Una prueba numérica del valor de una variable: Es $#argv más grande que 2?
- Una prueba del valor de la cadena de una variable: Contiene $confirma la cadena 'si'?
- Una prueba del tipo de un archivo: Es el archivo practica un directorio?
- Una prueba para ver si un archivo o variable existe?

La <condición> puede ser aún una combinación de dos o más pruebas condicionales. Si la condición se evalúa como cierta, la primera acción (uno o más comandos) después de la palabra then es realizada. Si la condición se evalúa como falsa, la acción que sigue a la palabra else es realizada. La cláusula else (la palabra else y la acción asociada) es opcional.

Las expresiones pueden ser usadas para efectuar varios tipos de pruebas y pueden tomar cualquiera de la siguientes formas:

- Para probar valores numéricos:
  Por ejemplo, num1 <operador> num2. El <operador> puede ser cualquiera de los siguientes, como en ($#argv == 0):

OPERADOR	FUNCIÓN
==	igual a
!=	diferente a
<	menor que
<=	menor o igual que
>	mayor que
>=	mayor o igual que

- Para probar valores de cadenas:

EXPRESIÓN	FUNCIÓN
cadena1 == cadena2	Verdadero si las dos cadenas son iguales
cadena1 != cadena2	Verdadero si las dos cadenas son diferentes
cadena1 =~ cadena2	Verdadero si la cadena1 concuerda el patrón de metacaracteres especificado por cadena2, como en ("dos palabras" =~ *p*)
cadena1 !~ cadena2	Verdadero si cadena1 no concuerda con el patrón de metacarateres especificado por cadena2, como en ("letras !~ ???)

- Para probar permisos, existencia, y tipo de un archivo como en (-d $archivo) para probar si $archivo es un directorio. También:

EXPRESION	FUNCION
-r nombrearchivo	Verdadero si el archivo tiene permisos de lectura.
-w nombrearchivo	Verdadero si el archivo tiene permisos de escritura.
-x nombrearchivo	Verdadero si el archivo tiene permisos de ejecución.
-e nombrearchivo	Verdadero si el archivo existe.
-o nombrearchivo	Verdadero si el usuario es el propietario del archivo.
-z nombrearchivo	Verdadero si el archivo está vacío (el tamaño es cero).
-f nombrearchivo	Verdadero si el archivo no es un directorio.
-d nombrearchivo	Verdadero si el archivo es un directorio.

Un lugar donde quizás se quieran realizar pruebas numéricas es adentro de cualquiera de los comandos de la mini agenda telefónica. Para hacer a los comandos más amigables al usuario, se le puede dar al usuario la opción de teclear el nombre y apellido como argumentos en la línea de comandos, o teclear simplemente el comando, en cuyo caso, se le pedirán posteriormente estos datos. Esto se ejemplifica a continuación:

```
#!/bin/csh
# 'Archivo: LOOKUP -- BUSCAR información en la agenda telefónica'
#
# 'Esta versión de LOOKUP le pregunta al usuario el nombre'
# 'de la persona a buscar si no se incluyo esta información'
# 'en la línea de comandos.'
#
clear
if ( $#argv < 1 ) then
   echo -n "Cual es el nombre de la persona a buscar? "
   set nombre=$<
   echo -n "Cual es el apellido de la persona a buscar? "
   set apellido=$<
   set nombrec="$nombre $apellido"
else
   set nombrec="$1 $2";      # 're-asignar argumentos de'
                             # 'la línea de comandos'
endif
grep "$nombrec" $HOME/datafile
if ( $status != 0 ) then      # 'si la persona no fue encontrada'
   echo "$nombrec no esta en tu base de datos."
endif
```

Observemos que el segundo if del ejemplo anterior hace uso de la variable $status. A esta variable se le asigna el valor de 0 (cero) si el comando inmediato anterior se ejecuto satisfactoriamente. En caso contrario se le asigna un valor numérico asociado al tipo de error que haya presentado el comando durante su ejecución.

Las comparaciones son también frecuentemente usadas para probar una respuesta de usuario a una pregunta en shell scripts interactivos. El siguiente shell script ilustra este uso así como también muestra una prueba para verificar la existencia de un archivo:

```
#!/bin/csh
# 'Archivo: previ -- preguntar al usuario por archivo de backup y'
#                   'entra al editor vi'
#
# 'Este comando checa si el archivo a editar ya existe.'
# 'Si si existe, el usuario tiene la opción de crear una'
# 'copia de backup del archivo antes de entrar al editor.'
#
if (-e $1) then          # 'si el archivo existe'
   echo -n "Quieres crear una copia de $1 (s/n)? "
   set respuesta=$<
   if ($respuesta == 's') then
      cp $1 "$1.bak";  # 'Crea el backup'
      echo "Una copia de $1 ha sido salvada en $1.bak."
   endif
endif
/usr/bin/vi $1
```

Sentencias if Anidadas.

Las sentencias if anidadas pueden ser confusas cuando son involucradas cláusulas else. Una regla importante que debemos recordar es que una cláusula else es siempre relacionada a la sentencia if más reciente.

Sentencias if/then/else Anidadas.

Frecuentemente es necesario especificar varias opciones en lugar de sólo una. En shell scripts interactivos, por ejemplo, se pueden usar varias sentencias if/then/else para checar la entrada y así detectar diferentes tipos de error. Cada condición es realmente usada para asociar una acción con un tipo especifico de entrada. Si alguna de estas condiciones es VERDADERA, ninguna de las restantes es probada.

El siguiente ejemplo muestra la estructura anidada para un genérico fragmento de código de checado de errores:

```
if (<condición error 1>) then
  echo <mensaje error 1>
else
  if (<condición error 2>) then
    echo <mensaje error 2>
  else
    if (<condición error 3>) then
      echo <mensaje error 3>
    else
      <entrada normal -- procesarla>
    endif
  endif
endif
```

Combinando Pruebas Condicionales.

Se puede lógicamente combinar pruebas condicionales así como efectuarlas separadamente. Hay tres tipos de operaciones lógicas: AND (&&), OR (||), y NOT (!). Sus efectos son resumidos en la tabla siguiente:

Operaciones Lógicas

OPERACIÓN	SINTAXIS	RESULTADO
NOT	!<expr>	Lo contrario del valor de <expr>. Si <expr> se evalúa como VERDADERO, !<expr> se evalúa como FALSO
OR	<expr1> || <expr2>	Se evalúa como VERDADERO si una o ambas expresiones se evalúan como VERDADERO
AND	<expr1> && <expr2>	Se evalúa como VERDADERO sólo si ambas expresiones se evalúan como VERDADERO

Precedencia de Operaciones Lógicas.

El operador AND (&&) tiene precedencia sobre el operador OR (||). El orden de las operaciones en la evaluación de sentencias condicionales es muy similar al orden de las operaciones en aritmética. El operador AND actúa como la multiplicación, y el operador OR actúa como la adición. Se pueden inclusive usar paréntesis para cambiar el orden de evaluación.

Ciclos

Como la mayoría de los lenguajes de programación, el shell también proporciona construcciones de flujo de control llamadas loops que nos permiten repetir secciones de código.

Loops Iterativos.

Para implementar un loop iterativo, se necesita la siguiente construcción:

```
foreach <variable-control-loop> ( <lista> )
        <lista de comandos>
end
```

Para la mayoría de los comandos que se escriban, quizás se quiera usar la lista de argumentos del comando para substituirlos por <lista>. Esta lista de argumentos es representada por *$argv[*]* en el C shell. Si se desea usar la lista de argumentos del comando, se debe usar explícitamente:

```
foreach <variable> ( $argv[*] )
```

El siguiente shell script permite teclear lf en lugar de *ls -F* para obtener una lista de archivos con directorios y archivos ejecutables específicamente marcados:

```
#!/bin/csh
# 'Archivo: lf -- lista archivos como la opción -F de ls'
#
# 'Este comando hace casi lo mismo que la opción -F'
# 'del comando ls. Su salida es una lista de los archivos'
# 'contenidos en los subdirectorios especificados en su línea'
# 'de argumentos.'
#
# 'Si una archivo es un subdirectorio, una diagonal (/) es'
# 'agregada al nombre de este, si es un archivo ejecutable,'
# 'entonces un asterisco (*) es agregado al nombre del archivo'
# 'Si no se dan argumentos, el directorio actual es listado.'
#
unset noclobber
if ( $#argv == 0 ) then  # 'lista el directorio actual'
                         # 'si no hay argumentos'
   ls -F
else
   foreach file ($argv[*])
      if (-f $file) then
         ls -F $file >> /tmp/${0}$$
      else
         echo "directorio: $file"
         ls -F $file
         echo " "
      endif
   end
   if (-e /tmp/${0}$$) then
     cat /tmp/${0}$$
     /bin/rm /tmp/${0}$$
   endif
endif
```

NOTA: Cuando la variable noclobber está definida, se recibe un mensaje de error si se trata de agregar una salida a un archivo que no existe usando redireccionamiento (>>). Si se coloca el comando *unset noclobber* al principio del script, este problema puede ser evitado.

Loops Condicionales.

Los loops condicionales son implementados usando sentencias while. La sintaxis es la siguiente:

```
while ( <condición> )
        <lista de comandos>
end
```

La condición es probada en la parte superior del loop. Si es verdadera, la lista de comandos (conocida como cuerpo del loop) es ejecutada. Típicamente, uno de los comandos en el cuerpo del loop cambiará el valor de prueba. El loop se continúa ejecutando mientras la condición sea verdadera.

Si se entra a un loop y ninguna de las sentencias dentro del loop cambia el resultado de la prueba de condición, el loop se continúa ejecutando infinitamente.

El siguiente script puede ser usado para implementar el comando CREATE.

NOTA: Antes de que este comando sea usado, el comando ADD debe ser modificado par checar la existencia de la base de datos.

```
#!/bin/csh
# 'Archivo: CREATE -- permite al usuario crear una'
# 'mini agenda telefónica'
#
clear
echo "Este comando permite crear una mini agenda telefónica."
echo "Toda la información necesaria le será pedida."
echo -n "Quieres crear una agenda telefónica? (s/n) ? "
set respuesta=$<
while ( $respuesta == 's' )
   ADD
   echo -n "Quieres agregar a alguien mas? (s/n) ? "
   set respuesta=$<
end
```

Otro uso común de los loops condicionales son los programas conducidos por menús. Este tipo de programas le dan al usuario una lista de opciones de la cual puede elegir. La elección es evaluada, el comando es ejecutado, y al usuario se le pide que seleccione otra opción. Cuando se escribe un programa de este tipo, hay que asegurar incluir una opción de ayuda (help) y una opción de salir (quit). La opción de ayuda vuelve a desplegar el menú en caso de que el usuario olvide los comandos. La opción de salir le permite al usuario salir del programa.

El siguiente shell script combina todas las operaciones para la mini agenda telefónica. Adicionalmente a los comandos ADD, DELETE, LOOKUP, y UPDATE, otras opciones han sido agregadas par permitirle al usuario ver el MENU de comandos, y salir (QUIT) del programa.

NOTA: El primer comando en el shell script es set noglob. Este comando es necesario para prevenir que el signo de interrogación sea interpretado como un metacaracter.

```
#!/bin/csh
# 'Archivo: info -- implementa la mini agenda telefónica'
# '             basada en un menú.'
#
clear
set noglob
echo -n "Bienvenido al programa de información de la "
echo "mini agenda telefónica."
set comando='?'
while ($comando != 'q' && $comando != 'Q')
    if ($comando == 'a' || $comando == 'A') then
        ADD
    else if ($comando == 'd' || $comando == 'D') then
        DELETE
    else if ($comando == 'l' || $comando == 'L') then
        LOOKUP
    else if ($comando == 'u' || $comando == 'U') then
        UPDATE
    else if ($comando == 'p' || $comando == 'P') then
        PRINT
    else if ($comando == '?') then
        MENU
    else
        echo "$comando no es un comando valido."
        echo "Teclea ? para ver una lista de comandos disponibles."
    endif
echo -n "Introduce un comando (? para ver una lista de opciones):"
    set comando=$<
end
```

```
#!/bin/csh
# 'Archivo: MENU -- imprime un menú de los comandos disponibles'
#
echo "Para ejecutar cualquier comando, teclea la primera letra del"
echo "nombre del comando.  Para ver la lista de comandos, teclea"
echo "un signo de interrogación (?)."
echo " "
echo "ADD          - agregar un nombre a la agenda telefónica"
echo "DELETE       - borrar un nombre de la agenda telefónica"
echo "LOOKUP       - buscar un nombre en la agenda telefónica"
echo "UPDATE       - actualizar información en la agenda telefónica"
echo "PRINT        - imprimir un listado de la agenda telefónica"
echo "QUIT   - salir del programa info"
echo " "
```

```
#!/bin/csh
# 'Archivo: UPDATE -- Actualizar la información en la'
# 'mini agenda telefónica'
# 'Este comando es usado para actualizar un registro en la'
# 'mini agenda telefónica contenida en el archivo datafile.'
# 'Cada registro consiste de un nombre y un apellido, seguido'
# 'de una fecha de nacimiento y de un número telefónico.'
# 'El usuario que corra este archivo de comandos le será'
# 'preguntada toda la información necesaria'

if ($#argv != 2) then
   echo -n "Cual es el nombre de la persona a actualizar? "
   set nombre=$<
   echo -n "Cual es el apellido de $nombre? "
   set apellido=$<
   set nombrec="$nombre $apellido"
else
   set nombrec="$1 $2"
endif
if { grep -sq "$nombrec" $HOME/datafile } then
   echo "El siguiente registro esta siendo actualizado:"
   grep "$nombrec" $HOME/datafile
   echo -n "Estas seguro de que lo quieres cambiar (s/n)? "
   set confirma=$<
   if ($confirma == 's') then
      grep -v "$nombrec" $HOME/datafile > /tmp/${0}$$
      echo "Teclea la nueva fecha de nacimiento (mes/día/año)"
      set dian=$<
      echo "Teclea el nuevo numero telefónico de $nombrec."
      set teléfono=$<
      echo "$nombrec<TAB><TAB>$dian<TAB><TAB>$teléfono" \
>> /tmp/${0}$$
      sort +1 -2 /tmp/${0}$$ > $HOME/datafile
      echo "El nuevo registro es:"
      echo "$nombrec<TAB><TAB>$dian<TAB><TAB>$teléfono"
   else
      echo "Tu base de datos no ha sido cambiada."
   endif
else
   echo "$nombrec no esta en tu base_de_datos."
   echo "Ejecuta el comando ADD para agregar a $nombrec."
endif
```

Accesando la Lista de Argumentos.

La mayoría de los comandos se utilizan con opciones o argumentos. Una de las funciones del shell es analizar gramaticalmente cada línea de comando o dividirla en sus componentes. Estas partes incluyen al comando, sus argumentos, y apuntadores de redirección así como sus archivos correspondientes. La primera cosa que el shell hace cuando interpreta un comando es estudiarlo. La redirección es tomada en cuenta en este momento. Después, las demás palabras en la línea de comandos son asignadas a variables con nombres numéricos, comenzando con cero para el comando mismo. Así que la línea de comando:

```
% grep -n "M" archivodatos personal amigos > mi.info
```

es divida de la siguiente forma:

```
$0 = grep    $3 = archivodatos
$1 = -n      $4 = personal
$2 = M       $5 = amigos
```

Estas variables son conocidas como parámetros posicionales porque sus nombres son derivados de sus posiciones en la línea de comandos. Estas le permiten a un shell script aceptar entrada en la línea de comandos. Sus valores son asignados por el shell cuando un comando es ejecutado, y pueden ser referenciados en el código del comando. Hay que notar que el apuntador de redirección así como el nombre de archivo asociado con él, no son parámetros posicionales.

Puede haber veces que se necesiten referir todos los argumentos del comando. En este caso hay que usar la variable $argv[*] (en el C shell). En el ejemplo anterior, el valor de $argv[*] es:

```
"-n M archivodatos personal amigos"
```

La variable $#argv, se refiere al número de parámetros posicionales (sin incluir el comando). En el ejemplo anterior, $#argv es 5.

Hay algunas limitaciones en el uso de parámetros posicionales. Sólo se puede hacer referencia a parámetros posicionales con nombres de dígitos sencillos ($1 - $9). Usualmente no se necesitan más de nueve parámetros posicionales, pero en caso de que se necesiten más, se pueden accesar. En C shell, se tiene que usar el comando *shift* para accesar a aquellos parámetros que su referencia sea mayor a la novena posición. El efecto del comando shift es el de desplazar la lista de parámetros un lugar hacia la izquierda, de manera que después de haber usado shift, el parámetro $1 es ahora substituido por el valor de $2, $2 es substituido por $3 y así sucesivamente. Si se volviese a hacer uso de shift, ahora $1 que fue substituido por $2 sería substituido por $3, $2 que fue substituido por $3 sería substituido por $4 y así como parámetros haya.

Los siguientes ejemplos consisten de un shell script llamado show, el cual imprime los valores de hasta seis parámetros posicionales:

```
% cat show
# /bin/csh
# 'Archivo: show -- muestra numeros y valores de parámetros'
# 'posicionales'
#
echo "El comando ("'$0'") es $0."
echo "El numero de parámetros ("'$#argv'") es $#argv"
echo '$1 es' $1
echo '$2 es' $2
echo '$3 es' $3
echo '$4 es' $4
echo '$5 es' $5
echo '$6 es' $6
%
% show un ejemplo con exactamente seis parámetros
El comando ($0) es show.
El numero de parámetros ($#argv) es 6
$1 es un
$2 es ejemplo
$3 es con
$4 es exactamente
$5 es seis
$6 es parámetros
%
```

% **show un ejemplo corto**
El comando ($0) es show.
El numero de parámetros ($#argv) es 3
$1 es un
$2 es ejemplo
$3 es corto
$4 es
$5 es
$6 es
%
% **show Cuando hay mas de seis parámetros los últimos \ son ignorados.**
El comando ($0) es show.
El numero de parámetros ($#argv) es 10
$1 es Cuando
$2 es hay
$3 es mas
$4 es de
$5 es seis
$6 es parámetros
%
% **show "Cualquier cosa entre comillas es" 'un solo parámetro'**
El comando ($0) es show.
El numero de parámetros ($#argv) es 2
$1 es Cualquier cosa entre comillas es
$2 es un solo parámetro
$3 es
$4 es
$5 es
$6 es

Si modificamos ahora un poco el comando show para que haga desplazamientos de variables utilizando shift quedaría algo parecido a lo siguiente:

```
# /bin/csh
# 'Archivo: show -- muestra numeros y valores de parámetros'
# 'posicionales'
#
echo "El comando (\"$0\") es $0."
echo "El numero de parámetros (\"$#argv\") es $#argv"
echo '$1 es' $1
echo '$2 es' $2
echo '$3 es' $3
shift; shift;           # 'Se hacen dos desplazamientos'
echo '$4 es' $4
echo '$5 es' $5
echo '$6 es' $6
%
% show un ejemplo con más de seis parámetros
El comando ($0) es show.
El numero de parámetros ($#argv) es 7
$1 es un
$2 es ejemplo
$3 es con
$4 es seis
$5 es parámetros
$6 es
```

Usando Variables de Ambiente.

Cuando se crean variables de ambiente se les puede dar casi cualquier nombre que se quiera. Hay algunos nombres, sin embargo, que representan variables de shell predefinidas. Estas variables son usadas por el shell para controlar el ambiente de trabajo. Un ejemplo de una variable de ambiente es el prompt del shell. El prompt en C shell está almacenado en la variable "prompt". Se puede cambiar el prompt, con un reset del valor de la variable apropiada:

```
% set prompt="<HOLA> "
<HOLA>
```

Hay que notar el espacio después de <HOLA>. Este espacio fue añadido para que al usuario le sea más fácil distinguir los comandos que teclee desde el prompt.

La siguiente tabla muestra algunas de las más importantes variables de ambiente en el shell. Así como se puede desplegar el valor de una variable utilizando sentencias echo. También se puede usar el comando *set* sin argumentos para desplegar los valores de TODAS las variables que están definidas.

Variables de Ambiente en el shell

NOMBRE VARIABLE	DESCRIPCION
cwd	Directorio actual de trabajo. Es definida por el shell cuando se ejecuta un comando cd.
history	Número de comandos a salvar en la lista de historia (history). En el C shell, también es el número a desplegar.
home	Directorio de entrada del usuario (home directory).
mail	Especifica el nombre del archivo que contiene el correo del usuario. Si la variable está definida. el shell checa aproximadamente cada 10 minutos si el archivo ha sido modificado desde la última vez que fue leído. Si es así, el mensaje "You have new mail" es impreso.
noclober	Si la variable está definida, el shell no permite sobrescribir archivos cuando se redirecciona la entrada usando el apuntador de redirección > . También previene crear archivos usando el apuntador de redirección >>. En cualquier caso se recibe un mensaje de error.
path	Contiene los nombres de los subdirectorios en donde se buscarán los comandos que se escriban desde la línea de comandos.
prompt	Valor del prompt primario del shell (el default es %, y # es usado para superusuario)
shell	Shell de entrada del usuario (intérprete de comandos). Por ejemplo: /usr/bin/ksh, /bin/csh, /bin/sh, o /usr/bin/sh5

Búsqueda de Comandos

Cuando se manda un comando para que el shell lo ejecute, se usa el valor de la variable PATH para encontrar el(los) directorio(s) donde el código para ese comando está localizado. Esto significa que cuando se le pasa un comando al shell, este busca si existe un archivo con ese nombre en el primer directorio listado en la variable PATH, si es así, lo ejecuta. Si no, busca en el siguiente directorio de la lista. Este proceso se repite hasta que se encuentra el comando o se termina la lista de subdirectorios. Si el comando nunca se encuentra, se recibe uno de estos dos mensajes de error:

<command name>: not found
<command name>: command not found

Por convención, los directorios de comandos en Unix son usualmente llamados bin. El nombre viene de la palabra binary y se deriva del hecho que las versiones ejecutables de la mayoría de los comandos de sistema están almacenadas en formato binario. La mayoría de los comandos de sistema están escritos en el lenguaje de programación C; la versión binaria es el estado ejecutable. Dentro de la cultura de Unix, las palabras binario y ejecutable han llegado a ser casi sinónimos.

Los usuarios y programadores de Unix usualmente almacenan sus comandos en un subdirectorio de su directorio hogar llamado bin. Después, redefiniendo el valor de la variable path para incluir el directorio de comandos propio, estos comandos se pueden ejecutar desde otros directorios.

Una de las ventajas de establecer el path con estos subdirectorios, es que se pueden escribir comandos con los mismos nombres que los comandos de sistema. Debido a que el directorio bin propio puede preceder a los directorios de sistema en la variable path, la versión propia del comando es ejecutada en lugar de la versión del sistema.

Características Especiales de las Variables de Shell

Las variables que son creadas dentro de un shell script sólo existen mientras el shell script se ejecuta. Las variables que están dentro de un shell script son llamadas variables locales, porque sus valores son locales al shell script que las usa. Si se trata de hacer referencia a ellas interactivamente, o si otro shell script trata de usarlas, éstas no existirán.

El shell trata a la sesión de terminal como si fuera un shell script. Las variables que se definen durante la sesión de terminal, incluyendo a las variables de ambiente, sólo están disponibles cuando se escriben comandos interactivamente. No se pueden modificar dentro de los shell scripts que se escriban.

```
% cat setpr
#!/bin/csh
# 'Archivo: setpr -- Muestra el valor de la variable prompt,'
# 'reasigna su valor y muestra el nuevo valor de la variable.'
#
# echo "El valor actual de la variable prompt es $prompt"
# echo -n "Escriba el nuevo prompt: "
# set prompt=$<
# echo "El nuevo valor de prompt es $prompt"
%
% setpr
El valor actual de la variable prompt es %
Escriba el nuevo prompt: LAA>
El nuevo valor de prompt es LAA>
%
```

Compartiendo Valores de Variables.

Algunas veces los comandos necesitan hacer referencia a los valores de ciertas variables de ambiente. El editor vi es un buen ejemplo de un comando así. Si el valor de la variable TERM no está definida, vi no opera como un editor de pantalla-completa.

Para compartir el valor de una variable, un shell *exporta* la variable. En C shell se usa el comando *setenv*. El valor de cualquier variable exportada es compartido por todos los procesos hijos del shell. Para exportar una variable, se debe usar la siguiente sintaxis:

% setenv <nombre variable> <valor>

En C shell, también hay un comando llamado printenv que imprime los valores de todas las variables que fueron definidas con el comando setenv. Adicionalmente, el comando printenv imprime algunas variables de ambiente que el C shell exporta automáticamente.

Se pueden exportar variables sólo al shell actual y a los *"hijos"* de ese shell. Ningún shell "padre" del shell actual se enterará de las variables que se exportaron.

Cabe señalar que cuando se encuentra un error en un shell script, éste se deja de ejecutar inmediatamente.

Usando Operaciones Aritméticas con Variables de Shell.

Adicionalmente a sus muchas otras capacidades, el shell puede efectuar aritmética de enteros simple. La aritmética puede ser usada en muchas formas diferentes dentro de los shell scripts. Uno de los usos más importantes es el conteo, el cual es frecuentemente usado para controlar la ejecución de los loops.

En C shell, el símbolo "*at*" (@) significa matemática. La siguiente tabla muestra como asignar los resultados de operaciones aritméticas a una variable llamada *numero*.

Operaciones Matemáticas

OPERACIÓN	ASIGNACIÓN VARIABLE
suma (a+b)	@ numero = a + b
resta (a-b)	@ numero = a - b
multiplicación (axb)	@ numero = a * b
división entera (parte entera de a/b)	@ numero = a / b
residuo de a/b	@ numero = a % b

El shell reconoce sólo enteros no negativos entre 0 y 32767. Si se trata de introducir un número negativo o un número con punto decimal en un shell script interactivo, se recibirá un mensaje de error. Los números más grandes que 32767 no son interpretados correctamente.

El siguiente ejemplo muestra el shell script countdown como un ejemplo de como la aritmética puede ser usada para controlar la
ejecución de un loop:

```
% cat countdown
# 'Archivo: countdown -- cuenta en reversa del 10 al 0'
#
set contador=10
while ($contador >= 0)
        echo "          $contador"
        @ contador=$contador - 1
end
echo "* F U E G O *"
%
% countdown
          10
          9
          8
          7
          6
          5
          4
          3
          2
          1
          0
* F U E G O *
%
```

Depurando los shell scripts

Si se encuentra un error con los datos de prueba probablemente se quiera efectuar una depuración ("debug") del programa siguiendo visualmente su ejecución línea por línea. Para seguir la ejecución de los shell scripts, se debe incluir el comando *set verbose* dentro de los scripts que provoque que el shell imprima cada línea del script según la ejecute.

El método más rápido de seguir la ejecución de un shell script es correrlo ya sea con la opción -v o -x del comando csh. Por ejemplo:

```
% csh -vx nombrearchivo
```

La opción -v imprime las líneas de entrada del shell según son leídas, mientras que la opción -x imprime los comandos y sus argumentos según son ejecutados.

Substituyendo Comandos.

Como ya se mencionó, se puede asignar la salida estándar de un comando a una variable con substitución de comandos. El formato general para la substitución de comandos es:

```
% set <variable>=<`comando`>
```

donde <`comando`> es un comando que escribe a la salida estándar.

Se puede usar substitución de comandos para evitar volver a teclear largas listas de palabras. Por ejemplo, quizás se quiera asignar los nombres de los archivos que contengan la cadena "include" a la variable incarchivos. El comando grep tiene una opción -l que imprime los nombres de los archivos que contienen palabras que concuerdan con la cadena dada en la línea de comandos.

```
% set incarchivos=$(grep -l 'include' *)
```

La variable incarchivos entonces contiene la lista de aquellos archivos que contienen la cadena de caracteres "include":

```
% echo $incarchivos
dosc1.c doc2.c fill.c h.c list.c large.c
```

En este momento se puede usar la variable incarchivos como entrada a otros comandos. Después, se puede ver el contenido de los archivos pantalla por pantalla, copiarlos a un directorio de backup, y mandarlos a la impresora:

```
% more $incarchivos
% cp $incarchivos /users/cecafi/amezcua/backup
% lp $incarchivos
```

La substitución de comandos no está limitada a la asignación de variables. Se puede usar la substitución de comandos para substituir argumentos en la línea de comandos. Por ejemplo, el comando cat se podría ejecutar así:

```
% cat `grep -l 'include' *`
```

FACULTAD DE INGENIERIA U.N.A.M.
DIVISION DE EDUCACION CONTINUA

SISTEMA OPERATIVO UNIX

PARTE III

PROGRAMACION EN KORN Y BOURNE SHELL

OCTUBRE 1994

Palacio de Minería Calle de Tacuba 5 Primer piso Deleg. Cuauhtémoc 06000 México, D.F. Tel.: 521-40-20 Apdo. Postal M-2285

Programación en Korn y Bourne Shell

El Bourne Shell es el interprete dce comandos estándar de UNIX. Fué desarroillado por Stephen Bourne en los laboratorios de AT&T, y actualmente es el único shell que forma parte de la configuración de cualquier UNIX.

El Korn Shell fué recientemente desarrollado, pero ha ido remplazando al Bourne Shell debido a que es completamente compatible con este y además incluye muchas características avanzadas con las que no cuenta el Bourne Shell.

El Bourne Shell, el Korn Shell y C Shell, son los tres shells más utilizados en los ambientes UNIX. Los dos primeros fuerón desarrollados para UNIX System V. El C Shell es comunmente utilizado en versiones BSD de UNIX. En el Release 4 de System V, los tres shells estan disponibles, debido a que esta versión de UNIX es una mezcla de las características de System V, BSD y XENIX.

1. Variables

Al igual que C shell, Korn Shell cuenta con una serie de variables de ambiente, las cuales pueden ser inicializadas al momento del login, desde la línea de comandos o desde los archivos de inicialización.

La siguiente tabla muestra las principales variables de Korn Shell, la columna tipo indica *login*, si la columna es inicializada automáticamente al momento de login; *shell*, si la variable es inicializada automáticamente por el shell; y *user*, si la variable es inicialmente no esta inicializada y el usuario la puede activar o desactivar.

NOMBRE	TIPO	DESCRIPCION
CDPATH		Directorios de búsqueda para el comando cd
EDITOR		Editor de comandos (emacs, ed, vi, etc.)
ERRNO	shell	Código de error de la última system call
HISTFILE		Archivo en donde se guarda la lista de comandos
HISTSIZE		Número de comandos en la lista de comandos
HOME	login	Argumento por default para el comando cd
IFS	shell	Separador de caracteres de entrada
LOGNAME	login	Login name
MAIL	login	Nombre del archivo de mail. Esta variable sirve para monitorear la llegada de mensajes
MAILCHECK	shell	Frecuencia en segundos de verificación de llegada de mensajes. El default es 600 segundos
MAILPATH		Semejante a MAIL, pero en esta se indican una lista de archivos a monitorear. La lista de archivos se separa con ":"
PATH	login	Lista de directorios de busqueda para la ejecución de comandos. La lista esta separada por ":"

PS1	shell	Prompt primario
PS2	shell	Prompt secundario, se utiliza, por ejemplo, cuando se teclea un comando en varias lineas. Por default es >.
PWD	shell	Directorio de trabajo actual
SECONDS	shell	Tiempo transcurrido en segundos desde el login
SHELL	login	Shell utilizado en la generación de subshells
TMOUT		Tiempo de inactividad en segundos, antes de terminar la sesión de shell automáticamente

Existen tres tipos de variables:

- regulares
- especiales
- arreglos

1.1 Variables regulares

Los identificadores de las variables regulares obedecen ciertas reglas:

- Debe de comenzar con una letra maúscula, minúscula o el caracter "_".
- Los caracteres restantes pueden ser cualquier número de letras (mayúsculas o minúsculas), digitos o el caracter "_".

Las variables regulares pueden ser locales o globales. Una variable local es conocida únicamente en el shell en donde es definida. Una variable global, también llamada exportada, es aquella que es conocida en el shell en donde se hizo su definición así como en los procesos hijos de este shell. Estos procesos no pueden cambiar el valor de la variable o destruirla.

Para asignar un valor a una variable especial se utiliza la siguiente sintaxis:

identificador=valor

no debe de existir blanco entre el identificador y "=". Si la variable no existe, esta se crea, si ya existe, simplemente se le cambia su valor.

Ejemplo:

```
$EDITOR=vi
$
```

Para hacer referencia a una variable, simplemente hay que preceder al identificador de la variable con el signo "$". Por ejemplo, para conocer el directorio HOME:

```
$echo $HOME
/usr/users/acf
$
```

Para delimitar el identificador de una variable o para controlar la sustitución de valores, se utilizan las siguientes expresiones:

${identificador}i	Reemplaza con el valor de la variable.
${identificador-word}	Reemplaza con el valor de la variable si esta definida, de otra forma con word.
${identificador+word}	Reemplaza con word si la variable esta definida, de otra forma con nada.
${identificador?word}	Reemplaza con el valor de la variable si esta definida, de otra forma se manda un mensaje de error con el formato: identificador: word
${identificador=word}	Si la variable no esta definida, se asigna el valor a la variable y el reemplazo se lleva a cabo con el nuevo valor.

Se pueden utilizar los operadores :-, :+, :? y :=, cuyo efecto es semejante al de la tabla, con la diferencia que en este caso se verifica la definición de la variable, o si esta tiene un valor nulo.

Ejemplos:

```
$cat > ${file - ${TMPDIR - /tmp/}/$LOGNAME}

$echo ${dir:-`awk -F: '$1 ==" 'LOGNAME' " {print $6}' `}

$cat *.txt > ${file=/usr/tmp/combined.txt}
$nroff $file | lpr
```

1.2 Variables especiales

Las variables especiales utilizan caracteres especiales para formar su identificador. Este tipo de variables no pueden ser definidas por el usuario y sus valores son asignados por el shell. La tabla siguiente muestra la lista de variables especiales.

$0	El nombre del comando.
$1 - $9	Argumentos en la línea de comandos.
$*	Todos los argumentos en la línea de comandos como "$1 $2 $3 ..."
$@	Todos los argumentos en la línea de comandos como "$1" "$2" "$3" ...
$#	El número de argumentos.
$?	El código de estatus del último comando ejecutado.
$$	El Process ID del shell actual.

La única forma de accesar los argumentos $9 en adelante es con ayuda del comando shift, el cual "rota" una o más posiciones la lista de argumentos en la línea de comandos:

```
$set uno dos tres cuatro cinco seis siete ocho nueve diez
$ echo $# $*
10 uno dos tres cuatro cinco seis siete ocho nueve diez
$shift
$echo $# $9
9 diez
$
```

1.3 Variables de ambiente

Las variables de ambiente, son aquellas que tienen asignado un valor constante y que son globales. Las variables de ambiente pueden ser referenciadas desde cualquier proceso hijo del shell que la crea, sin que estos procesos tengan que ser necesariamente shells.

La forma de definir variables de ambiente es la siguiente:

```
$UNIXDIR = $HOME/unix
$export UNIXDIR
```

La forma de conocer las variables definidas en el shell, así como sus valores asociados, es con el comando set:

```
$ set
CDPATH=:/users/acf
HOME=/users/acf
IFS=
LOGNAME=acf
MAIL=/usr/spool/mail/acf
PATH=/bin:/usr/bin:/usr/local/bin:.:/users/acf/bin
PS1=$
PS2=>
SHELL=/bin/sh
$
```

Para conocer únicamente la lista de variables de ambiente se puede utilizar el comando env o printenv:

```
$env
CDPATH=:/users/acf
HOME=/users/acf
LOGNAME=acf
MAIL=/usr/spool/mail/acf
PATH=/bin:/usr/bin:/usr/local/bin:.:/users/acf/bin
$
```

2. Control de procesos

UNIX es un sistema operativa multitarea en donde aparentemente se pueden ejecutar varios procesos al mismo tiempo.

Decimos que aparentemente, ya que el sistema operativo solamente atiende a un procesos en pequeños intervalos de tiempo; sin embargo la rapidez con que atiende a cada uno de los procesos da la impresión de que todos los procesos se ejecutan al mismo tiempo.

En UNIX se puede tener cierto control sobre los procesos del usuario, es decir se puede monitorear su desempeño, controlar el tiempo de CPU que utiliza, suspender o terminar un proceso, etc.

2.1 ¿Qué es un proceso?

Un proceso consiste de un espacio en memoria y un conjunto de estructuras de datos que lo identifican.

El espacio en memoria que el sistema operativo marca para uso exclusivo de un proceso contiene espacio para el código del programa, para las variables que el proceso utiliza y para el stack.

Las estructuras de datos, que son almacenadas en el núcleo del sistema operativo, contienen información como:

- El mapa de direcciones de memoria.

- El estado.

- La prioridad de ejecución.

- Registro de los recursos utilizados.

- El dueño del proceso.

2.2 Parámetros de un proceso

Muchos de los parámetros asociados a un proceso afectan directamente la ejecución de este: archivos que puede accesar, salida de datos, etc. estos se encuentran almacenados ya sea en el kernel en las estructuras de datos o bien en su espacio de memoria. A continuación se explican los más importantes.

Identificador de Proceso (PID)

El identificador de proceso (PID, *Process IDentification*) es un número único que el núcleo asigna a un proceso para su identificación. El número es asignado conforme los procesos se van creando.

Identificador del Proceso Padre (PPID)

El Identificador del Proceso Padre (PPID, *Parent Process IDentification*) es el PID del proceso que creó el proceso en cuestión.

Identificador del Usuario (UID)

El Identificador del Usuario (UID, *User IDentification*) es un número con el que se identifica al usuario que creo el proceso. Este usuario es el dueño del proceso y es el único que puede cambiar los parámetros de este.

Efectivo Identificador de Usuario (EUID)

El Efectivo Identificador de Usuario (EUID, *Effective User IDentification*) es el UID utilizado para determinar los recursos que puede accesar el proceso.

Prioridad

La prioridad de un proceso determina el tiempo de CPU que dicho proceso puede utilizar.

El CPU determina el siguiente proceso a ejecutarse en base a las prioridades.

Para sistemas BSD las prioridades de un proceso están en el rango +19 a -19 y en sistemas ATT están en 40 a -20, en donde las prioridades más altas son las negativas.

Todos los procesos de usuario tienen un prioridad por omisión. El dueño de un proceso solo puede disminuir la prioridad de este

Un proceso solo puede cambiar su prioridad de acuerdo a como lo pueda su dueño.

Un proceso hereda la prioridad del proceso que lo crea.

Terminal asociada

Todos los procesos en UNIX tienen asociada una terminal desde la cual se llevan a cabo las operaciones de entrada y hacia la cual se dirigen las salidas del proceso (esto a menos que haya redireccionamiento).

Estados de un proceso

Debido a que un proceso no se encuentra en ejecución en todo momento, se definen cinco posibles estado para un proceso:

Ejecución: un proceso en ejecución es un proceso que ha adquirido tiempo

de CPU en ese instante, así como también todos los recursos necesarios para entrar en operación.

Sleeping: un proceso en estado sleeping es aquel que espera la ocurrencia de un evento; mientras tanto, el proceso no puede solicitar tiempo de CPU.

Swapp: los procesos en estado de Swapp son aquellos que han sido transladados de memoria a disco. Este estado de un proceso se genera principalmente en sistemas que no tienen memoria virtual.

Zombie: los procesos Zombie son procesos que no tienen nada; pero por alguna razón el kernel guarda información de ellos.

Stop: un proceso en estado de Stop es aquel que ha sido marcado por el núcleo para que no pueda entrar en ejecución a menos que se le mande una señal de continuación. Los procesos son enviados a este estado cuando se les manda una señal de STOP.

2.3 Manejo de señales

Las señales son el medio por el cual se pueden controlar los procesos o bien, se puede llevar a cabo una comunicación entre ellos.

Cuando un proceso recibe una señal, pueden suceder dos cosas: si el proceso cuenta con una rutina o manejador para esa señal, esta es ejecutada; de otra forma, el núcleo proporciona un manejador para esta señal.

Cuando existe una manejador para una señal se dice que la señal se "atrapa".

Para prevenir que deliberadamente se envíen señales a un proceso, este puede ignorar o bloquear ciertas señales. Existen dos señales que no pueden ser atrapadas, ignoradas o bloqueadas: KILL y STOP.

Existen 17 señales estándar para versiones AT&T y 31 para sistemas BSD, cada una de ellas tiene asociado un identificador y un nombre simbólico.

Número	Nombre	Descripción
2	SIGINT	Interrupción
3	SIGQUIT	Terminación
4	SIGILL	Instrucción ilegal
9	SIGKILL	Destrucción
10	SIGBUS	Error de bus
12	SIGSYS	Llamada al sistema con argumentos no válidos
14	SIGALARM	Alarma

Señales estándar para sistemas AT&T

Número	Nombre	Descripción
2	SIGINT	Interrupción
3	SIGQUIT	Terminación
4	SIGILL	Instrucción ilegal
9	SIGKILL	Destrucción
10	SIGBUS	Error de bus
12	SIGSYS	Llamada al sistema con argumentos no válidos
14	SIGALARM	Alarma
17	SIGSTOP	Detención
19	SIGCONT	Continuación
20	SIGCHLD	Estado del proceso hijo ha cambiado
25	SIGXFSZ	Tamaño de archivo excedido
29	SIGLOST	Recurso dañado

Señales estándar para sistemas BSD

2.4 Envío de señales

La comunicación entre procesos se puede llevar a cabo por medio del envió de señales.

El comando **kill** permite enviar una señal a un proceso en particular.

Para enviar una señal a un proceso, es necesario ser el dueño de dicho proceso, solamente el superusuario puede mandar señales a cualquier proceso.

La sintaxis del comando kill es la siguiente:

kill [-señal] PID

La señal se puede especificar por su número o nombre, PID es el Identificador del Proceso al que se le envía la señal. Si no especifica señal, se envía la señal 15 (TERM).

2.5 Modificación de la prioridad de un proceso

La prioridad de un proceso puede ser cambiada para aumentar o disminuir el tiempo de CPU que recibe.

Los usuarios normales solamente pueden decrementar la prioridad de sus procesos, el superusuario es quien puede cambiar la prioridad de cualquier proceso en cualquier sentido.

En sistemas BSD la prioridad de un proceso puede determinarse al momento de crear el proceso, con ayuda del comando **nice**, el cuál tiene la siguiente sintaxis:

nice [nivel] comando [argumentos]

Si no se especifica un nivel de prioridad se crea un proceso con prioridad 10.

Si un nivel de prioridad se especifica, este se añade el nivel por omisión para determinar el nuevo nivel de prioridad.

El nivel de prioridad por omisión es 0. Por ejemplo, para incrementar la prioridad a un proceso se deberá especificar un nivel negativo:

```
% nice -10 comando
%
```

Cuando se trabaja con csh, el comportamiento de nice varía un poco, en este caso el comando es opcional y el nivel de prioridad es interpretado relativo al nivel del proceso padre.

Cuando no se especifica proceso en el comando nice, se cambia el nivel de prioridad para el proceso de shell actual (por lo tanto para sus procesos hijos).

Cuando no se especifica un nivel de prioridad, se incrementa en 4.

En sistemas BSD, la prioridad de un proceso también puede cambiarse una vez que este se ha creado; esto se logra con ayuda del comando **renice**. La sintaxis para el comando es la siguiente:

renice nivel [-p PID ...] [-g pgroup] [-u usuario]

El nivel especifica el nuevo nivel de prioridad. Con la opción **p** se especifican los PID's de los procesos afectados, la opción **g** sirve para indicar un grupo, con lo cuál se afectan los procesos que pertenecen al grupo especificado. Por último la opción **u** permite listar el usuario para el cual sus procesos serán afectados.

En sistemas AT&T, el comportamiento del comando nice es diferente. En este caso el nivel de prioridad se interpreta de acuerdo al nivel de prioridad actual del proceso. Por ejemplo:

% nice -5 nomina
%

incrementa el nivel de prioridad en 5.

Si no se especifica nivel, este se incrementa en 10. Solamente el superusuario puede bajar el nivel de prioridad (en realidad se incrementa la prioridad de los procesos):

% nice --10 nomina

Cuando se especifican niveles de prioridad de -100 o menos, los procesos se ejecutan en tiempo real, es decir se les asignan todo el tiempo de CPU.

2.6 Procesos en paralelo

Cada vez que se proporciona un comando al shell, este lo interpreta, lo ejecuta y cuando termina devuelve el control al usuario para que este pueda proporcionar un segundo comando.

En realidad cada vez que se proporciona un comando al shell, este una vez que lo interpreta crea un proceso hijo para ejecutar el comando.

UNIX es un sistema operativo multitarea, por lo que se pueden mandar a ejecutar procesos al mismo tiempo (aunque en realidad no sucede así).

La forma de crear procesos en paralelo es con el terminador &.

Cuando se finaliza un comando con &, el shell crea un proceso hijo como es usual, pero no espera a que este termine, en lugar de ello, el shell muestra el PID del nuevo proceso y de inmediato muestra el indicador para otro comando:

```
% nomina &
2764
%
```

Si el proceso en paralelo requiere de alguna entrada, se le envía una señal de STOP.

La salida generada por el proceso en paralelo siempre es enviada a la salida estándar mientras el usuario que arranco el proceso continué en sesión.

Cuando el usuario que inició el proceso paralelo termina su sesión de UNIX, el proceso paralelo (hijo del shell de inicio del usuario) es adoptado por el proceso *init* (cuyo PID es 1).

Cuando es adoptado el proceso, este sigue conservando todos sus parámetros, salvo que ya no tendrá una terminal asociada y su PPID será 1. Por otra parte, la salida generada por este proceso se perderá.

En sistemas AT&T cuando un shell es terminado se envía una señal de HUP a todos sus procesos hijos, incluyendo los que fueron ejecutados en paralelo.

Si se desea que un proceso paralelo continué después de que termina el shell desde el cuál se ejecutó, se deberá utilizar el comando **nohup** de la siguiente forma:

nohup comando &

Este comando ignora la señal NOHUP.

Cuando los comandos se ejecutan en la manera usual, esto es, esperando a que éstos terminen para poder dar el siguiente comando, se dice que se están ejecutando en *foreground.* Estos comandos que se ejecutan en foreground pueden ser detenidos mediante otros comandos o bien, mediante teclas de control.

En cambio, los comandos que al ejecutarse no esperan a terminar para regresarle al usuario el control del prompt, se dice que son comandos en background a los cuales se les puede hacer referencia por su número de trabajo o 'job_number'.

Es posible ver cuantos comandos están corriendo en background utilizando el comando jobs, el cual nos dará un reporte del estado actual del trabajo así como el comando que se mandó a ejecutar.

El comando jobs lista los trabajos por su número de trabajo (job number), el cual es usado para detenerlo (**stop**), continuarlo (**bg** o **fg**) o bien, para terminarlo (**kill**).

La siguiente tabla muestra los comandos para control de trabajos.

COMANDO	FUNCION
comando &	Comienza a ejecutar ese comando o trabajo en background.
jobs	Lista por número de trabajo, los comandos que se encuentran corriendo en background o que están suspendidos.
bg *%job_number*	Ocasiona que el comando referenciado por *job_number* continúe su ejecución en background.
fg *%job_number*	Ocasiona que el comando referenciado por *job_number* que se encuentra corriendo en background o que se encuentra suspendido, sea ahora ejecutado en foreground.
^Z	Suspende un comando que se encuentra corriendo en foreground.
stop *%job_number*	Suspende al comando referenciado por *job_number*.
kill *-sig %job_number*	Manda la señal *sig* al trabajo *job_number*.
kill *-sig* PID	Manda la señal *sig* al proceso referenciado por PID.
kill -l	Lista las señales que pueden ser mandadas a los procesos o a los trabajos.

2.7 Monitoreo de procesos

El comando **ps** permite obtener información para los procesos existentes en el sistema.

El comando ps incluye opciones, las cuales pueden variar de sistema a sistema.

% ps

```
PID  TTY      TIME COMMAND
5643 console  0:02 -csh
6214 console  0:00 ps
```

% ps -elf

```
F S   UID  PID  PPID  C PRI NI   ADDR  SZ   WCHAN   STIME TTY      TIME COMD
3 S  root    0     0  0 128 20 40013000   0  126ccc  Dec 31 ?      0:00 swapper
1 S  root    1     0  0 168 20 40013180  41  904000  Aug  3 ?      0:00 /etc/init
3 S  root    2     0  0 128 20 400130c0   0  146e74  Aug  3 ?      0:00 vhand
3 S  root    3     0  0 128 20 40013100   0  11e0cc  Aug  3 ?      0:00 statdaemon
3 S  root  200     0  0 128 20 40013140   0  146e7c  Aug  3 ?      0:00 unhashdaemon
3 S  root    6     0  0 152 20 40013240   0 4003aa2c  Aug  3 ?      0:00 sockregd
3 S  root    4     0  0 154 20 40013280   0  11eec0  Aug  3 ?      0:00 lcsp
3 S  root  201     0  0 152 20 400132c0   0  11e0e8  Aug  3 ?      0:00 syncdaemon
1 S  root 5643     1  3 168 20 400524c0  46  904000 10:51:00 console  0:02 -csh
1 S  root  609     1  0 154 20 400139c0  13  126cdc  Aug  3 ?      0:00 /etc/syslogd
1 S  root  958     1  0 154 20 400521c0  17  147c77a Aug  3 ?      0:00 /etc/cron
1 S  root  241     1  0 127 20 40013640  19  111234  Aug  3 ?      0:00 /etc/nktl_daemon
1 S  root  604     1  0 154 20 40013880  22 40044f22 Aug  3 ?      0:00 /etc/rlbdaemon
1 S  root  244   243  0 127 20 400134c0 129 1476060  Aug  3 ?      0:00 netfmt -CF
1 S  root  614     1  0 154 20 40013b40  34  126cdc  Aug  3 ?      0:00 /etc/portmap
1 S  root  617     1  0 154 20 40013ac0  20  126cdc  Aug  3 ?      0:00 /etc/inetd
1 S  root  620     1  0 154 20 40013c40   8 40049e2c Aug  3 ?      0:30 /etc/rwhod
1 S  root  636     1  0 154 20 400523c0  11 40049a22  Aug  3 ?      0:00 /usr/bin/nftdaemon
1 S  root  960     1  0 155 20 40052540  12  126dbc  Aug  3 ?      0:00 /etc/ptydaemon
1 S  root  643     1  0 154 20 40052500  41  126cdc  Aug  3 ?      0:00 /etc/snmpd
1 S  root  963     1  0 154 20 400525c0  16  126cdc  Aug  3 ?      0:00 /etc/vtdaemon
1 R  root 6226  5643  8 180 20 40052f00  16        16:40:45 console  0:00 ps -elf
3 S  root 5882     0  0 153 20 40052a40   0  11eed2 16:29:01 ?      0:00 gcsp
```

La información que se presenta es, entre otra:

Estado

S	Durmiendo (Sleeping)
W	En espera (Waiting)
R	En ejecución (Running)
Z	Terminado (Terminated)
T	Detenido (Stopped)
X	Creciendo (Growing)

UID

PID

PPID

3. Control de flujo

3.1 if

Un if es en si una sentencia condicional.

Su sintaxis completa es la siguiente:

```
if list [ ; ]
then list [ ; ]
[elif list [ ; ] ] ..
[else list [ ; ] ]
fi
```

Sin embargo comenzaremos simplificandola un poco:

```
if comando1
then comando2
else comando3
fi
```

La parte *else* es opcional. Cuando el shell encuentra una expresión if del tipo de la anterior, ejecuta las siguientes acciones:

- Ejecuta el comando1
- Si el comando1 regresa un estatus de 0 (indicando éxito), se ejecuta el comando2.
- Si el estatus es diferente de 0, se ejecuta el comando3.
- El estatus de la expresión if será TRUE si el estatus del último comando ejecutado fué 0 de otra forma FALSE.

Ejemplo:

```
$ if test - d $HOME
> then echo $HOME es un directorio
> else echo $HOME no es directorio
> fi
/usr/acf es un directorio
$
```

En la sintáxis completa del if, se observa que, opcionalmente se pueden indicar listas de comandos *list* en el if. La lista de comandos es una secuencia de comandos separados por el caracter ";" o, por el carcater nueva línea.

En el caso en el que se incluyan listas de comandos, el estatus del último comando determina el estatus de la lista, que permitira la evaluación del if.

Ejemplo:

```
if diff $FILE1 $FILE2 > /tmp/diff.out
then                                    # Los archivos son iguales
        echo "$FILE1 : $FILE2 son archivos iguales"
        rm $FILE2 /tmp/diff.out
else                                    # Los archivos no son iguales
        pg /tmp/diff.out
        rm -i $FILE1 $FILE2
        rm /tmp/diff.out
fi
```

En construcciones anidadas, la parte *else* termina el if más interno, no se toma en cuenta el sangrado.

Una decisión múltiple puede implementarse con una serie de if anidados; sin embargo, el sangrar cada una de las sentencias provocaría que el tamaño de la línea creciera demasiado, para ello se emplea una construcción como la siguiente:

```
if comando
then   comando
elif comando
then   comando
elif  comando
then     comando
elif comando
then   comando
elif comando
else comando
fi
```

En la construcción anterior, las expresiones se evalúan en orden, cuando alguna de ellas es verdadera, la sentencia asociada se ejecuta y con esto se termina la

construcción. La sentencia del último else se ejecuta cuando ninguna expresión es verdadera.

El comando if puede ser escrito en varias líneas, o bien se puede agrupar en una sola; en este caso, cada una de las sentencias deberá ser separada por el caracter ";". Por ejemplo:

```
if cmp -s $FILE1 $FILE2; then echo "Son iguales"; else echo "son diferentes; fi
```

para mejor comprension del if, la palabra then puede ser colocada en la primera línea, colocando el caracter ";" como separador, por ejemplo:

```
if cmp -s $FILE1 $FILE2> /tmp/diff.out; then
        cat $FILE1
else
        cat /tmp/diff.out
fi
```

La construcción if puede ser tratada completamente como un comando, de tal forma, que la salida de este se puede redireccionar, o bien se puede evaluar el código de estatus de esta construción. Por ejemplo:

```
if cmp -s $FILE1 $FILE2> /tmp/diff.out; then
        cat $FILE1
else
        cat /tmp/diff.out
fi > res.out
```

La construción if es muy utilizada en conjunto con el comando test. Para aumentar la legibilidad de los scripts de Korn shell, el comando test se escribe en forma diferente a la forma tradicional *test expresion*, esto es [*expresion*].

El comando test es utilizado para checar la existencia de un archivo, para verificar las características de este, para evaluar el valor de una variable, para comparar dos cadenas, etc.

Ejemplo:

```
if [ -f $FILENAME ]
then echo "El archivo existe"
else echo "Archivo inexistente"
fi >&2
```

Para validar el que una variable tenga valor:

```
if [ -z "$FILENAME" ] ; then
        echo "El nombre delarchivo debe ser especificado" ; exit
fi
```

Para verificatr el valor de una variable:

```
if [ "$USER" = "acf" ] ; then
        ...
else
        echo "$USER: acceso restringido" >&2
        exit
fi
```

En la siguiente tabla se hace un resumen de las expresiones que se pueden utilizar en el comando test.

EXPRESION	SIGNIFICADO
-b file	El archivo existe y es un block-special file.
-c file	El archivo existe y es un caracter-special file.
-d file	El archivo existe y es directorio.
-f file	El archivo existe y es un archivo regular.
-g file	El archivo existe y tiene activado el set-group-ID bit.
-h file	El archivo existe y es una liga.
-h string	El string no es nulo.
-r file	El archivo existe y tiene permiso de lectura.
-w file	El archivo existe y tiene permiso de escritura.
-x file	El archivo existe y es ejecutable.
-z string	El string es nulo.
-s file	El archivo existe y su tamaño es mayor a cero bytes.
-t [num]	El file descriptor num esta asociado a una terminal.
-u file	El archivo existe y tiene activado el set-user-ID bit.
string	El string es no nulo.
a = b	El string a es igual a b.
a != b	El string a es diferente del string b.
a -eq b	a es númericamente igual a b.
a -ne b	a es númericamente diferente a b.
a -gt b	a es númericamente mayor a b.
a -ge b	a es númericamente mayor o igual a b.
a -lt b	a es númericamente menor a b.
a -le b	a es númericamente menor o igual a b.
file1 -nt file2	El archivo file1 es más reciente que el archivo file2.
file1 -ot file2	El archivo file1 es menos reciente que el archivo file2.
file1 -ef file2	Los archivos file1 y file2 son el mismo archivo.

3.2 case

La proposición switch permite la implementación de decisiones múltiples con expresiones de tipo string.

Sintaxis:

case *string-expresion* in
[*pattern* [| *pattern*] ... *command-list* ;;] ...
esac

donde:

string-expresion = expresión de tipo string

pattern = uno o varios patrones de tipo string separados por "|"
la lista de patrones debe finalizar con ")"

command-list = lista de comandos

Los patrones tipo string pueden contener metacaracteres.

El último comando de la lista de comandos debe terminar con ";;"

La expresión se evalúa y el resultado se compara con los patrones; si alguno de ellos coincide, se ejecuta la lista de comandos asociada y termina la sentencia case.

Los patrones no se deben repetir.

El siguiente ejemplo lee un string de la entrada estandar y verifica que comience con "y":

```
read cad
case $cad in
        y*) ;;
        *) exit 1 ;;
esac
```

El patron * sirve como claúsula por default.

Ejemplo:

```
if [ ! -d $DIR ] ; then
        case $DIR in
        . | ..) echo "No se puede crear . o .." >& 2 ;;
        *)
          if mkdir $DIR ; then
                chmod 0765 $DIR
          else
                echo "No se puede crear $DIR" >& 2
          fi ;;
        esac
fi
```

3.3 for

Sintaxis:

for *var* [in *words*]
do
    list
done

for *var* [in *words* ;] do *list* ; done

words es una lista de strings separados por blancos, los strings pueden ser valores de variables, la salida de un comando, expresiones con metacaracteres, nombres de archivos, etc. Si se omite esta lista de strings, también se deberá omitir la palabra in.

Cuando se omite la lista de strings se asume por default que su valor es $*, esto es, la lista de argumentos en la línea de comandos.

var es el nombre de una variable, no su valor ($var).

list es una lista de comandos que es ejecutada una vez por cada palabra en *words*, en cada ciclo, *var* adquiere el valor de cada palabra. La lista de palabras es procesada de izquierda a derecha.

El siguiente ejemplo manda un mensaje a una lista de usuarios:

```
list=${1:- /etc/passwd}
cat > ${MSGFILE=${TMPDIR:-/tmp/msg.$$}}
for USER in `cut -d: -f1 < ${list} | sort -u`
do
        mail $USER < $MSGFILE && echo "Usuario $USER notificado" >&2
done
rm $MSGFILE
```

3.4 while

Sintaxis:

```
while list1
do
        list2
done
```

```
while list1 ; do list2 ; done
```

La lista de comandos *list2* es ejecutada mientras la evaluación de la lista de comandos *list1* sea verdadera, esto es mientras el código de status del último comando de la lista *list1* sea cero.

3.5 until

Sintaxis:

```
until list1
do
        list2
done
```

until *list1* ; do *list2* ; done

La sentencia until es opuesta a la sentencia while. Se lleva a cabo la lista de comandos *list2* mientras el estatus de la lista de comandos *list1* sea falsa.

El siguiente ejemplo monitorea la una sesión de un usuario:

```
until who | grep acf > /dev/null
do
        sleep 60
done
echo "acf ha entrado a sesión"
```

3.6 break

Sintaxis

break [*n*]

Un break causa una salida inmediata de las siguientes construcciones:

- while
- for
- until

Opcionalmente se le puede inidicar a la sentencia break mediante *n* el número de ciclos que se cerrarán, por default es uno.

Ejemplo:

```
while :
do
        echo "Se borrara el directorio $DIR (S/N)? "
        read answer
        case $answer in
          Y*) rmdir $DIR ; break ;;
          N*) echo "se mantiene $DIR" ; break ;;
          *)   echo "Proporcione N o Y" ;;
        esac
done
```

3.7 continue

Sintaxis

continue[*n*]

Un continue provoca que el shell abandone la ejecución del resto de sentencias en un ciclo de for, while o until y comience un ciclo nuevo.

Opcionalmente se le puede inidicar a la sentencia continue mediante *n* el ciclo que se cancelará, esto para cuando se tienen ciclos anidados, por default es uno.