Se permite la distribucion por cualquier medio de acuerdo con la licencia GPL v.2 o posteriores. Puede obtener una copia de la licencia GPL o ponerse en contacto con la Free Software Foundation en http://www.gnu.org/

Programacion Shell

Adrian de los Santos
demon@demon.com.mx

Version 0.002b

ATENCION:

Este es un documento no terminado y sin revisiones, su fucncion fue la de servir como material de apoyo en un curso que imparti, no es una guia definitiva de la programacion shell (ni se le acerca por lo menos a un intento de guia).

Le faltan mucho por explicar, y algunas cosas que posiblemente no esten explicadas de forma correcta.

Como ayudar ?

Requiero tu retroalimentacion, enviame un correo a demon@demon.com.mx con tus comentarios acerca de este documento, que te parecio, que le cambiarias, que le falto, etc.

Esa es la mejor forma en la que podemos mejorar este documento.

Gracias.

Adrian de los Santos.

Indice

Introduccion
La programacion en shell
Comandos
Los basicos del shell
Caracteristicas del shell
Algunos comandos basicos
Inicializacion
Substitucion e interpretacion
Control de entrada/salida
Variables
Control de procesos
Filtros de texto
Sed
Awk
Control de flujo
Entrada y salida de datos en programas de shell
Expect

INTRODUCCION

No es un secreto que los sistemas operativos Unix/Linux han evolucionado en los ultimos años como un sistema operativo popular. Para los programadores que han utilizado Unix/Linux por muchos años, esto no es una sorpresa: Los sistemas Unix/Linux proveen una plataforma eficiente y elegante para el desarollo de sistemas. Despues de todo, esto es lo que Dennis Ritchie y Ken Thompson buscaban cuando ellos desarollaron Unix en los laboratorios Bell (a finales de los 60's).

Una de las caracteristicas fuertes de los sistemas Unix/Linux es su gran coleccion de programas. Mas de 200 comandos basicos que se incluyen con el sistema operativo. Estos comandos, (tambien conocidos como herramientas) hacen practicamente todo, desde contar el numero de lineas en un archivo, enviar correo electronico, desplegar un calendario de el año deseado, etc.

Pero la real fortaleza de los sistemas Unix viene no precisamente de esta gran coleccion de comandos, sino tambien de la elegancia y facilidad con que estos comandos pueden ser combinados para realizar funciones mas sofisticadas.

Con el fin de proveer una interface consistente y facil para que el usuario interactuara con el sistema Unix/Linux (el kernel) se desarollo el shell.

El shell es simplemente un programa que lee los comandos que se teclean y los convierte en una forma mas entendible para el sistema Unix/Linux. Tambien incluye algunas sentencias basicas de programacion que permiten: tomar desiciones, realizar ciclos y almacenar valores en variables.

El shell estandar distribuido con Unix y Linux, se deriva de la distribucion de AT&T, el cual a su vez, evoluciono de una version originalmente escrita por Stephen Bourne en los laboratorios Bell. Desde entonces la IEEE ha creado estandares basados en el Bourne Shell y otros shells mas recientes. La version actual de este estandar es "The Shell and Utilities Volume of IEEE Std 1003.1-2001", tambien conocido como es estandar POSIX (Portable Operating System Unix)

Debido a que el shell ofrece un lenguaje de programacion interpretado, se pueden escribir, modificar y verificar programas rapidamente y de forma facil.

La programacion en shell es una parte fundamental de la administracion de sistemas basados en Unix, debido a la facilidad y poderio que el conjunto de herramientas y comandos de Unix proveen para realizar la automatizacion de procesos rutinarios, tales como: respaldo de archivos, captura de datos, verificacion de procesos, etc.

Generalidades

En este curso se asume que estas familiarizado con los fundamentos de los sistemas Unix/Linux; esto es, que sabes como entrar a el sistema, como crear archivos, editarlos, manipularlos y como trabajar con directorios.

Dentro de este manual existe codigo de ejemplo, el cual se ejemplifica con un tipo de letra diferente, el siguiente es un ejemplo de un segmento de codigo:

```
$ ls -la
```

La programacion en shell

El programar en shell es muy similar a un oficio comun, por ejemplo: un carpintero.

Un carpintero tiene una caja de herramientas que contiene todas las cosas que utiliza para su oficio, en esa caja puede haber desarmadores, tornillos, taladros, etc. El carpintero utiliza estas herramientas de diferente manera y en diferentes combinaciones para lograr resultados diferentes, no se utiliza la misma herramienta para hacer un jugete que para hacer un escritorio y es posible que si se utilice la misma no se utilice en la misma intensidad o forma.

Aplicando estas mismas herramientas, el carpintero es capaz de construir los diferentes elementos necesarios para construir sus proyectos.

Para constuir algun objeto de madera, se necesitan las herramientas correctas. En Unix, las herramientas que se utilizan son llamadas "utilerias" o "comandos". Existen comandos simples como **`ls`** y **`cd`**, y existen herramientas mas complejas como **`awk`**, **`sed`**, y el mismo shell. Uno de los problemas mas comunes de trabajar con madera, es la de utilizar la herramienta o tecnica incorrecta para construir algun proyecto. El saber que herramienta utilizar, normalmente se obtiene con la experiencia. En este curso aprenderas como utilizar las herrameintas de Unix por medio de ejemplos y ejercicios. Las herramientas simples, son faciles de entender y aplicar. Es posible que tu ya conozcas varias de estas herramientas. Las herramientas mas poderosas, normalmente toman mas tiempo para entender y aprovechar.

En este curso introduce el uso de herramientas basicas y complejas. Logicamente la focalizacion es el utilizar las herramientas mas complicadas y mas poderosas (tal como el shell mismo).

Antes de que se puedan construir cosas con el shell, se necesitan conocer algunas cuestiones basicas:

- Comandos
- El Shell

Comandos

Que es un comando ?

En Unix, un "comando" es un programa que tu puedes ejecutar. En otros sistemas operativos, tales como Mac OS o Windows, you apuntas a el programa que deseas ejecutar y realizas la funcion de dar doble click sobre el. Para ejecutar un comando en Unix, tu teclas su nombre y presionas Enter.

Por ejemplo:

```
$ date [Enter]
Fri Mar 19 21:34:59 CST 2004
$
```

Al introducir este comando, **date** despliega el nombre del dia, mes, numero de dia transcurrido del mes, hora, zona horaria y año de el sistema.

Hay que hacer notar que despues de ejecutar el comando, el sistema despliega el caracter **$**

El caracter **$** indica el prompt (o linea de comando). Cuando se esta en el prompt, se puede teclear el nombre de un comando y presionar Enter. Esto ejecuta el comando que se tecleo. Mientras el comando esta siendo ejecutado, el prompt (**$**) no es desplegado en la pantalla. Cuando el comando finaliza su ejecucion, el prompt es desplegado nuevamente.

Hay que hacer notar que el caracter $ es el indicativo de que el sistema esta listo para recibir un comando nuevo, esto comunmente se llama prompt. No es parte del comando mismo.

Por ejemplo, para ejecutar el comando **date**, se teclea la palabra **date** en el prompt **$**. No se teclea **$ date**.

Ahora veamos otro ejemplo de un comando:

```
$ who
demon   tty1    Mar 16 02:26
root    tty2    Mar 16 01:26
tatito  pts/1   Mar 14 00:23
$
```

Aqui, se introdujo el comando who, el cual despliega los usuarios que se encuentran dentro del sistema, la terminal desde la cual estan conectados y la fecha en la que entraron al sistema.

Aqui podemos ver que existen 3 usuarios dentro del sistema, demon, root y tatito, la primera columna muestra el nombre de los usuarios, la segunda columna la terminal desde la cual el usuario entro al sistema y la tercera la fecha y hora en la que entraron a este sistema.

El formato y valores de despliegue de cada comando puede variar ligeramente de un sistema Unix a otro, o de un sistema Linux a otro (con diferentes versiones de kernel o diferentes distribuciones).

Comandos Simples

Los comandos **who** y **date** son ejemlos de comandos simples. Un comando simple es el que puedes ejecutar simplemente teclando su nombre en el prompt de la siguiente manera:

```
$ comando
```

En este ejemplo, comando es el nombre de el comando que deseas ejecutar. Los comandos simples pueden ser comandos pequeños y con funciones especificas y sencillas, tales como who y date, o pueden ser comandos largos, tales como un navegador de web o un programa de hoja de calculo. Tu puedes ejecutar la mayoria de los comandos de Unix como comandos simples.

Comandos

Comandos complejos

Puedes usar el comando who para obtener informacion acerca de tu usuario (que esta dentro del sistema) cuando se ejecuta de la siguiente manera:

```
$ who am i
demon    ttyp5    Mar 19 22:16
$
```

Este comando despliega la siguiente informacion:

- Mi nombre de usuario es demon
- Estoy entrando desde la terminal ttyp5
- Entre a las 22:16 del 3 de Marzo

Este comando tambien nos sirve para introducir el concepto de comando complejo, lo cual es un comando que consiste en un comando (valga la redundancia) y una lista de argumentos.

Los argumentos son modificadores que cambian el comportamiento de un comando. En este caso, el nombre del comando es **who** y los argumentos **am** e **i**.

Cuando el comando **who** se ejecuta como un comando simple, despliega informacion acerca de todos los usuarios que estan en el sistema. La salida que es generada cuando un comando es ejecutado como un comando simple es llamado el comportamiento por defecto de ese comando (default behavior).

Los argumentos **am** e **i** cambian el comportamiento del comando **who** para listar informacion acerca de ti solamente. En Unix/Linux, la mayoria de los comandos aceptan argumentos que modifican su comportamiento.

La sintaxis formal para un comando complejo:

```
$ comando argumento1 argumento2 argumento3 ... argumentoN
```

Aqui, **comando** es el nombre del comando que deseas ejecutar, y **argumento1** hasta **argumentoN** son los argumentos que deseas proporcionar al comando.

Comandos compuestos

Una de las caracteristicas de los sistemas *nix es la capacidad de combinar comandos simples y complejos con el fin de obtener comandos compuestos.

Un comando compuesto consiste en una lista de comandos simples y complejos separados por el caracter de punto y coma (;). Un ejemplo de un comando complejo es:

```
$ date ; who am i
Fri Mar 19 22:39:17 CST 2004
demon    ttyp5    Mar 19 22:39
$
```

Aqui el comando compuesto, consiste en el comando simple **date** y el comando complejo **who am i**. Como puedes ver por los datos desplegados el comando **date** es ejecutado primero y despues el comando **who am i**. Cuando se introduce un comando complejo, cada uno de los comandos individuales que componen al comando complejo se ejecutan en el orden en que fueron tecleados.

En el ejemplo anterior, el comando compuesto se comporta como si se hubieran tecleado los dos comandos por separado por ejemplo:

```
$ date
Fri Mar 19 22:39:17 CST 2004
$ who am i
demon    ttyp5    Mar 19 22:39
$
```

La diferencia basica entre ejecutar comandos de esta manera y ejecutarlos de forma separada es que con un comando compuesto no se obtiene el prompt entre cada uno de los comandos que se ejecuta.

La sintaxis formar para un comando compuesto es:

```
$ comando1 ; comando2 ; comando3 ; ...  ; comandoN
```

Aqui, desde **comando1** hasta **comandoN** pueden ser comandos simples o complejos. El orden de ejecucion es: Primero se ejecuta **comando1**, despues **comando2**, despues **comando3** y asi consecutivamente. Cuando **comandoN** termina su ejecucion el prompt regresa.

Separadores de comandos

El caracter punto y coma (;) es tratado como un separador de comandos, lo cual indica "donde un programa termina, el otro comienza"

Si no se usa para separar cada uno de los comandos individuales en un comando compuesto, la computadora no sabra donde termina un comando y donde comienza el otro. Por ejemplo, si ejecutamos el ejemplo anterior sin el punto y coma (;) obtendremos:

```
$ date who am i
date: illegal time format
usage: date [-nu] [-r seconds] [+format]
       date [[[[[cc]yy]mm]dd]hh]mm[.ss]
$
```

Asi es, un bonito error.

Porque ?

debido a que el comando **date** piensa que esta siendo ejecutado como un comando complejo con los argumentos **who**, **am** e **i**. El comando **date** no entiende esos argumentos y despliega el mensaje de error.

Tambien se puede utilizar el punto y coma para indicar la terminacion individual de comandos simples y complejos, por ejemplo:

```
$ who am i;
demon    ttyp5    Mar 19 22:56
$ who am i
demon    ttyp5    Mar 19 22:56
$
```

En el primer caso, se ejecuto who am i; (notar el punto y coma al final), en el segundo caso, se ejecuto who am i sin punto y coma al final, el resultado del comando es exactamente el mismo.

En multiples lenguajes de programacion (Pascal, C, Perl, Ruby, etc), se utiliza el caracter de punto y coma para designar el final de la linea, y muchos programadores de shell scripts lo utilizan de la misma forma, mas como una cuestion de costumbre que funcional.

Los basicos del shell

Que es el Shell ?

En la seccion anterior se explico que cuando se teclea el comando

`$ date`

El sistema ejecuta el comando date y despliega el resultado

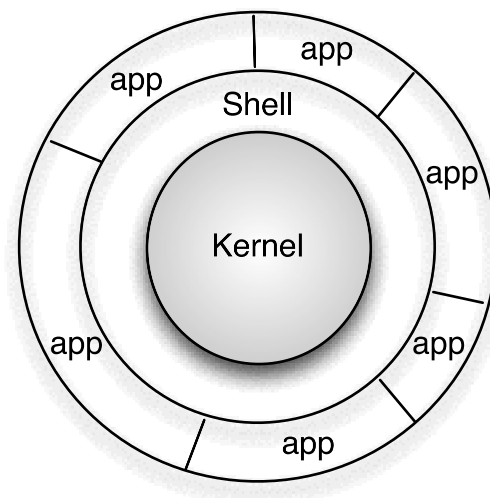
Pero como sabe el sistema que tu deseas ejecutar el comando date ?

El sistema utiliza un programa especial llamado *shell* para realizar esto. El shell provee una interface a el sistema. Obtiene la informacion del usuario y ejecuta programas en base a esa informacion. Cuando el programa finaliza su ejecucion, despliega la salida de el programa.

Por esta razon el shell es conocido como el interprete de comandos. Para usuarios familiarizados con Windows o DOS, el shell es similar a el archivo **`command.com`**.

Para definirlo de una manera simple, el shell es un programa que ejecuta programas (Aunque realmente es mucho mas que eso).

Una de las grandes ventajas de los sistemas *nix es que el shell es mas que un interprete de comandos, tambien es un lenguaje de programacion completo, con instrucciones de condicionales, asignacion, ciclos y funciones.



Los basicos del shell

El prompt del shell

El indicador **$** (o prompt) que vimos en las paginas anteriores es generado por el shell. (en algunos sistemas este indicador incluye tu nombre de usuario, el nombre de la maquina y el directorio en el que te encuentras, eso varia de sistema a sistema)

Mientras que el prompt esta desplegado, tu puedes introducir comandos. El shell lee la entrada despues de presionar Enter. Determina el comando a ejecutar examinando la primera palabra de los datos introducidos. Una palabra es un conjunto de caracteres continuos. Los espacios y tabuladores separan las palabras.

Para el shell los datos introducidos son analizados de la siguiente manera:

$ palabra1 palabra2 palabra3 ... palabraN

El shell siempre eligira **palabra1** como el nombre de el comando que deseas ejecutar. Si solo existe una palabra tal como:

$ date

El trabajo del shell es facil, ejecuta el comando date, pero si existen mas palabras, tal como:

$ who am i

El shell pasa las siguientes palabras como parametros a el comando indicado por la primera palabra (en este caso, el comando seria **who** y los parametros **am** e **i**).

Los basicos del shell

Los diferentes tipos de Shells

Desde la creacion de el sistema Unix original han existido diferentes tipos de shells con diferentes caracteristicas, el primer shell (el original creado por Dennis Ritchie y Ken Thompson) unicamente era un interprete de comandos, localizaba el comando y lo ejecutaba, cuando el comando terminaba el control regresaba a el shell, el cual estaba listo nuevamente para recibir algun otro comando o instruccion.

Durante los mas de 30 años que los sistemas *nix han existido se han creado una gran variedad de shells con diferentes caracteristicas, la mayoria de estos shells estan basados en 2 tipos de shell's:

Bourne Shell (**sh**).- El shell mas popular de los sistemas *nix, por mucho tiempo fue la unica opcion existente para interactuar con el sistema. Este shell ha sido la base para crear diferentes variantes del mismo, algunas de estas variantes son:

-Korn Shell (**ksh**)
-Bash o Bourne Again Shell (**bash**)
-POSIX shell (**sh**)

C shell (csh).- Creado por Bill Joy (co-fundador de Sun y creador de Java) a finales de los 70's. De igual manera que el Bourne shell permite ejecutar comandos, pero adiciona algunas caracteristicas de funcionalidad, tal como un historial de comandos y mejor control de procesos. Este shell ha sido la base para la creacion de las siguientes variantes:

- TENEX/TOPS C shell(tcsh)

Cual shell es mejor ? Bueno esa es una pregunta casi religiosa para algunas personas. El mejor shell es con el que eres mas productivo.

Durante este curso utilizaremos el Bourne Again Shell (**bash**) debido a que es el shell de default para los sistemas Linux, adicionalmente el 99% de la programacion en **bash** es totalmente compatible con Bourne shell, el cual esta disponible en cualquier sistema *nix, significando que se podra aplicar lo aprendido en este curso en la mayoria de los sistemas Unix existentes.

Los basicos del shell

El Bourne Shell (sh)

El shell original de Unix fue creado a mediados de los 70's por Stephen R. Bourne mientras el trabajaba en los laboratorios Bell AT&T de New Jersey. El Bourne shell fue el primer shell que aparecio en los sistemas Unix. El Bourne shell se encuentra normalmente en el directorio **bin** y su nombre de archivo es **sh** (**/bin/sh**). En adicion a ser un interprete de comandos el Bourne shell es un lenguaje de programacion con una estructura similar a el antiguo lenguaje de programacion **ALGOL**.

El Bourne shell contiene las siguiente caracteristicas:

- Control de procesos
- Variables
- Expresiones regulares
- Control de flujo
- Control de Entrada/Salida
- Soporte a Funciones

Pero algunos de los problemas de el Bourne shell original son:

- No existe soporte a autocompletado de nombres de archivos (el uso del tabulador)
- No existe historial de comandos ni edicion en linea
- Dificultad para ejecutar multiples procesos en background

Es posible que muchas de estas caracteristicas y desventajas del Bourne shell no se lleguen a comprender por completo en este momento, prero como vayamos avanzando en el curso tocaremos nuevamente estos topicos.

Caracteristicas del shell

El Bourne Again Shell (bash)

El Bourne Again Shell, **bash**, fue desarollado como parte del proyecto GNU y ha reemplazado a el Bourne shell, **sh**, en los sistemas basados en GNU, tal como Linux. Practicamente todas las distribuciones de Linux utilizan **bash** como su reemplazo de **sh**.

Incluye caracteristicas de C shell, Korn Shell y logicamente Bourne shell.

bash fue escrito inicialmente por Brian Fox (bfox@gnu.ai.mit.edu) y es actualmente mantenido por Chester Ramey (chet@ins.cwru.edu).

Debido a que **bash** es totalmente compatible con el Bourne shell, la mayoria de las distribuciones de Linux reemplazan el Bourne shell con **bash** (**/bin/bash** y **/bin/sh** son el mismo archivo)

Algunas de las caracteristicas de el Bourne Again Shell son:

- Autocompletado de nombres de variables , usuarios, nombres de servidores, comandos y archivos
- Correccion de rutas de archivos en el comando cd
- Soporte a arrays de tamaño ilimitado
- Aritmetica de enteros en cualquier base numerica (entre 2 y 64)

Caracteristicas del shell

Resumen

En las paginas anteriores hemos visto algunos aspectos basicos del shell, tales como: ejecucion de comandos simples, complejos y compuestos.

Adicionalmente se vio una breve introduccion a la historia y diferentes tipos de shell's existentes en los sistemas *nix.

Falta mucho mas que decir del shell, ayudame y dime que le falta aqui!.

Algunos comandos basicos

Algunos comandos basicos que utilizaremos mas adelante

Listando archivos:

ls.- su nombre se deriva de LiSt, es decir, lista los nombres de los archivos que se encuentran en el directorio indicado, si no se le indica ningun directorio, lista los archivos del directorio en el cual se encuentra el usuario.

Ejemplo:

ls [Enter]

Despliega los nombres de los archivos en el directorio actual.

Desplegando el contenido de un archivo:

cat.- El nombre **cat** viene de su funcion, concatena y despliega el contenido de la entrada estandar (la cual puede ser un archivo), el parametro que utiliza comunmente es el nombre del archivo a desplegar.

Ejemplo:

cat /etc/hosts [Enter]

Despliega el contenido del archivo **/etc/hosts**

Desplegando texto:

echo.- Este es uno de los comandos mas faciles de utilizar, simplemente despliega los argumentos que se introducen, por ejemplo:

```
$ echo hola
hola
```

En el ejemplo anterior el comando **echo** recibe como argumento la palabra **hola**, el comando unicamente despliega el argumento que se le introdujo, por lo cual la salida del comando es **hola**.

```
$ echo hola    como     estas
hola como estas
```

En el ejemplo anterior se introducen 3 argumentos, hay que notar los espacios existentes entre cada argumento y la salida del comando. El comando **echo** utiliza los espacios y tabulaciones para distinguir un parametro de otro, por lo cual no se respetan los espacios existentes entre los parametros.

```
$ echo "hola    como    estas"
hola    como    estas
```

En este caso se le esta especificando a el comando **echo** que **hola   como   estas** es un solo parametro y no 3 parametros separados, para hacer esto se encierra el parametro entre comillas ("). Y como podemos ver ahora los espacios entre las palabras si son respetados.

Copiando archivos:

cp.- El nombre del comando tambien nos dice su funcion, **cp** viene de la palabra en ingles "copy", por lo cual este comando copia archivos, recibe dos parametros, el primero es que archivo copiar y el segundo a donde copiarlo.

Ejemplos:

```
$ cp /etc/hosts /tmp
```

Este comando copiara el archivo llamado hosts que se encuentra en el directorio /etc a el directorio /tmp (en caso de que el directorio /tmp exista, de lo contrario creara un archivo llamado tmp en el directorio principal con una copia del contenido del archivo /etc/hosts)

```
$ cp archivo1.txt respaldo.txt
```

En este caso se copia el archivo con nombre **archivo1.txt** a otro archivo. El archivo de copia tendra el nombre **respaldo.txt**.

Como podemos ver, en el comando **cp** los parametros pueden ser archivos o directorios.

Contando palabras:

wc.- El nombre del comando viene de la palabra en ingles "word count" o contar palabras, aunque no solo sirve para contar palabras, tambien lineas y caracteres. Podemos usar **wc** para contar las palabras que existen en un archivo, por ejemplo:

```
$ wc /etc/hosts
      17      86     567 /etc/hosts
```

El primer numero en este caso es el numero de lineas del archivo, el segundo es el numero de palabras, el tercero el numero de caracteres y el cuarto el nombre del archivo. El nombre del archivo es importante cuando se especifica mas de un archivo en el comando:

```
$ wc /etc/hosts /etc/passwd
      17      86     567 /etc/hosts
      16      72     722 /etc/passwd
      33     158    1289 total
```

Si se especifica mas de un archivo, **wc** despliega los resultados de cada archivo individual y el total de todos los archivos.

Se puede utilizar el comando **wc** para obtener resultados individuales (lineas, palabras o caracteres) especificando las siguientes opciones:

Opcion	Descripcion
-l	Cuenta el numero de lineas
-w	Cuenta el numero de palabras
-m o -c	Cuenta el numero de caracteres

De esta manera:

```
$ wc -l /etc/hosts
      17 /etc/hosts
```

Despliega unicamente el numero de lineas existentes en el archivo /etc/hosts.

Desplegando los procesos del sistema

ps.- Su nombre significa Process Status, permite desplegar los procesos que estan siendo ejecutados en el sistema; cuenta con multiples opciones, pero para fines practicos solo veremos algunas de estas.

Para desplegar los procesos que estan siendo ejecutados por el usuario simplemente se teclea ps y enter en la terminal:

```
$ ps
PID  TT  STAT      TIME COMMAND
  394  p1  Ss+    0:00.21 -csh
  423  p2  Ss+    0:00.08 -csh
 6239 std  Ss     0:00.01 -bash

$
```

El comando **ps** muestra los procesos que el usuario esta ejecutando en ese momento, en este caso son 3 procesos.

Se muestra en 3 columnas, la primera es el numero de proceso (un numero consecutivo que se le asigna a cada proceso creado) o Process ID (**PID**), la segunta columna es la terminal desde la cual fue ejecutada el proceso (**TT**), la tercera columna es el status del proceso (si esta en el procesador, en espera de datos, si esta detenido, etc.), la cuarta columna es el tiempo consumido de cpu y la ultima es el comando realizado.

Para mostrar una lista de todos los procesos del sistema con las mayores caracteristicas tecleamos:

```
$ ps -ef
```

Los parametros -ef le indican a el proceso ps que deseamos una lista entera (**entire**) y completa (**full**) de los procesos del sistema. Este comando mostrara todos los procesos que el sistema tiene en ese momento, incluso los procesos que no son de nosotros.

Algunos comandos basicos

Mostrar un archivo una pantalla a la vez

Una de las caracteristicas del comando **cat** es la de poder desplegar en la pantalla el contenido de un archivo especifico. Uno de los problemas de el comando cat es que si el archivo es muy extenso, el principio del archivo pasara rapidamente por la pantalla y no seremos capaces de verlo, es decir que solo veremos las ultimas 24 lineas del archivo.

Para resolver este problema existe el comando **more**, este comando permite despelgar un archivo (o la entrada de datos que se le suministre) una pantalla a la vez, es decir: de 24 lineas en 24 lineas.

```
$ more /etc/passwd
```

Si el archivo **/etc/passwd** contiene mas de 24 lineas, se mostraran las primeras 24 lineas, y se esperara entrada de datos por parte del usuario para indicar la accion que se debera tomar a continuacion, si el usuario presiona **Enter**, entonces se despliegara una linea mas, si el usuario presiona **Espacio**, se despliegaran otras 24 lineas.

Nota: Tu mejor amigo se llama "man", utiliza el comando man para saber mas de los comandos anteriores y conocer mas opciones de los mismos, por ejemplo:

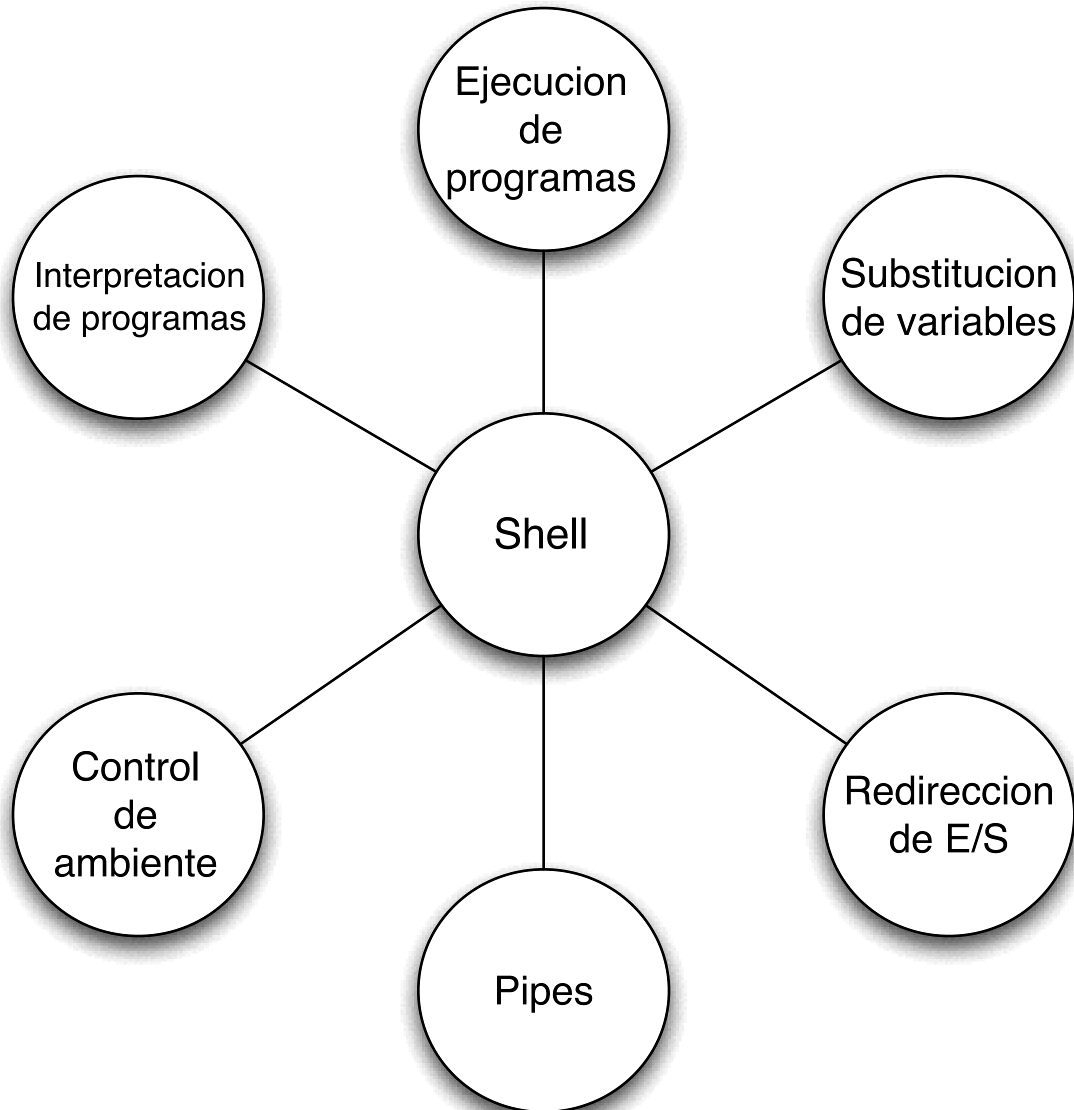
teclea:

```
$ man echo
```

Y podras conocer mucho mas de la forma de operacion del comando echo

Esto no es opcional, **TIENES QUE HACERLO**, leer el manual del comando te ayudara muchisimo cuando hagas programas en shell.

Las responsabilidades del shell



Inicializacion

Una terminal esta conectada a el sistema a traves de cable serial, telefoninco (modem) o red. Tradicionalmente en la pantalla de la terminal (o emulador de terminal) aparece la palabra **login:**.

Para cada terminal habilitada existe un programa llamado **getty** que activa la funcion de login. (en el caso de logins o accesos a traves de la red, no se utiliza **getty**, se utiliza el proceso o demonio "**inetd**", el cual habilita el servicio ya sea de **telnet**, **rlogin** o **ssh**)

El sistema, mas bien un programa del sistema llamado **init** automaticamente ejecuta **getty** en cada terminal en la cual se permite que los usuarios puedan entrar. **getty** determina la velocidad de transmision , despliega el mensaje **login:** y entonces espera que el usuario introduzca datos, tradicionalmente el nombre del usuario, despues de presionar Enter, getty despliega el mensaje **password:** en la terminal y espera que el usuario introduzca su contraseña. Una vez obtenido el nombre de usuario y su password, estos se comparan con la entrada correspondiente en el archivo **/etc/password**. Este archivo contiene una linea por cada usuario del sistema. Esta linea especifica entre otras cosas el nombre de usuario, el directorio de casa (**home directory**) y que programa se ejecutara una vez que el usuario entre al sistema. En este ultimo segmento de informacion (el programa que se ejecuta cuando el usuario entra al sistema) indica el shell que se ejecutara cuando el nombre de usuario y su password esten verificados correctamente.

Cuando el shell se ejecuta, realiza las siguientes funciones:

- Si existe el archivo **/etc/profile**, ejecuta las instrucciones que se encuentren en el.
- Si existe el archivo **.profile** en el directorio de casa del usuario se ejecutan las instrucciones que se encuentran en el.
- Despliega el prompt de el shell (tipicamente un signo de pesos **$**) y espera que se introduzcan comandos

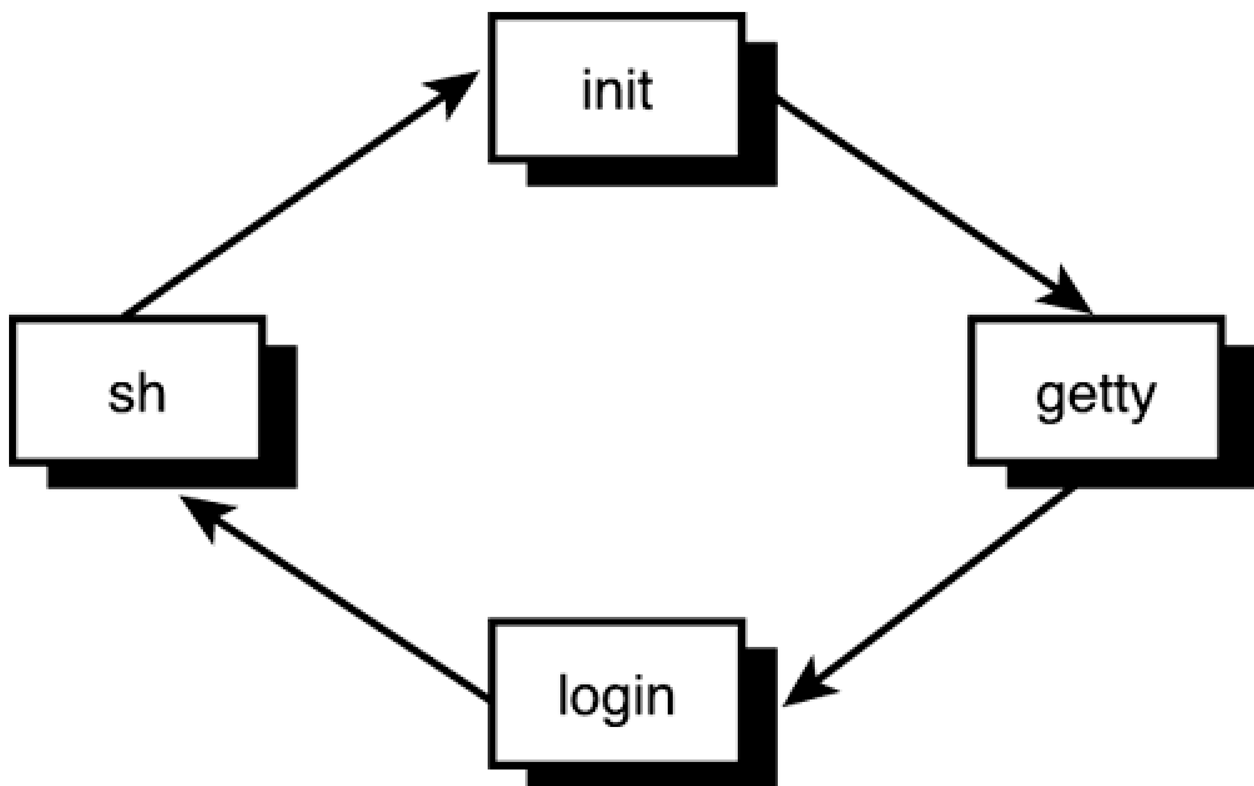
Cada vez que se introduce un comando y se presiona la tecla **Enter**, el shell analiza la linea y procede a realizar el requerimiento solicitado. Si se le pide que ejecute un programa en particular, el shell busca el programa en ciertas partes del disco y si lo encuentra le pide a el kernel que inicie la ejecucion del programa, el shell entonces "duerme" hasta que el programa ha finalizado. El kernel copia el programa especificado en la memoria y comienza su ejecucion. Esta copia del programa en la memoria es llamado proceso; de esta manera se puede definir que **un programa es el que esta almacenado en un archivo en el disco, mientras que un proceso se encuentra en la memoria ejecutandose.**

Si el programa envia su salida a traves de la salida estandar (**standard output**), esta aparece en la terminal a menos de que esta salida sea redirigida o "**piped**" a otro comando. De forma similar, si el programa lee la entrada de la entrada estandar

(**`standard input`**) entonces el programa esperara que se tecleen datos en la terminal a menos que se le envie la entrada por medio de un "**`pipe`**" o se le rediriga de un archivo.

Cuando el programa finaliza su ejecucion el control regresa nuevamente a el shell.

El ciclo de login



Es importante reconocier que el shell es unicamente un programa, no tiene privilegios especiales en el sistema, lo cual significa que cualquier persona con suficiente tiempo, conocimiento y devocion puede crear su propio shell. Esta es una de las razones por las cuales existen tantas variantes de shells para sistemas *nix.

Substitucion e Interpretacion

Substitucion e interpretacion

Una de las funcionalidades mas utilizadas del shell es la substitucion de nombres de archivos.

Supongamos que tenemos un directorio con los siguientes archivos:

```
$ ls
archivo1
archivo2
archivo3
$
```

Digamos que se desea desplegar el contenido de cada uno de estos archivos. Podriamos tomar ventaja de que el comando **cat** permite especificar mas de un archivo a la vez. Cuando esto se hace, los contenidos de los archivos son desplegados uno despues de otro:

```
$ cat archivo1 archivo2 archivo3
```

pero tambien podriamos teclear

```
$ cat *
```

Y obtener el mismo resultado. El shell automaticamente substituye los nombres de todos los archivos en el directorio actual por el caracter *. La misma substitucion ocurre si utilizamos el comando **echo**:

```
$ echo *
archivo1 archivo2 archivo3
$
```

Como hemos visto, el caracter * es reemplazado por el nombre de todos los archivos que se encuentran el el directorio actual, y el comando **echo**, simplemente despliega estos nombres en la terminal.

En cualquier lugar en el que aparezca el caracter * el shell realiza esta substitucion:

```
$ echo * : *
archivo1 archivo2 archivo3 : archivo1 archivo2 archivo3
$
```

Substitucion e Interpretacion

El ***** tambien puede ser usado en combinacion con otros caracteres para limitar los archivos por los cuales sera substituido. Por ejemplo, digamos que en el directorio actual no solo existen los archivos: **archivo1**, **archivo2** y **archivo3**, sino tambien los archivos **prog1**, **basura** y **trabajo**:

```
$ ls
archivo1
archivo2
archivo3
prog1
basura
trabajo
$
```

Para desplegar los nombres de archivos que comiencen con los **archiv**, tecleamos:

```
$ ls archiv*
archivo1
archivo2
archivo3
$
```

El caracter especial ***** no solo esta permitido al final del archivo; tambien puede ser utilizado al principio o en la mitad de el requerimiento, por ejemplo:

```
$ echo *vo1
archivo1
$ echo *chi*
archivo1
archivo2
archivo3
$ echo *x
```

En el primer ejmplo ***vo1** es reemplazado por todos los nombres de archivos que terminen con los caracteres **vo1**.

En el segundo ejemplo ***chi*** reemplaza todos los archivos que en cualquier parte del nombre contenga los caracteres "**chi**"

Y en el tercer ejemplo, y tomando en cuenta que no existe ningun archivo en el directorio que termine con el caracter **x**, que se desplegara ?

Substitucion e Interpretacion

Comodines para caracteres

El asterisco (*****) es un comodin para uno o mas caracteres, lo cual significa que **x*** se expande como **x1**, **x2**, **xabc** y asi sucesivamente. El signo de interrogacion (**?**) es un comodin para un solo caracter. De esta manera el comando **cat ?** desplegara el contenido de todos los archivos en el cual su nombre sea de un solo caracter y **cat x?** desplegara el contenido de todos los archivos en el cual su nombre sea de dos caracteres y el primer caracter sea **x**.

```
$ ls
a
aa
aaa
aab
bb
b
cc
a1
ccca
ccaa
c
archivo1
archivo2

$ echo ?
a
b
c

$ echo ??
aa
bb
cc
a1

$ echo a?
aa
a1

$ echo ??*
aa aaa aab bb cc a1 ccca ccaa archivo1 archivo2
```

Substitucion e Interpretacion

En el ejemplo anterior, **??** coincide con dos caracteres, y el ***** coincide con cero o mas caracteres. El efecto real es que **??*** coincide con todos los nombres de archivos que tengan dos o mas caracteres

Otra forma de buscar coincidencias con un solo caracter es la de crear una lista de caracteres para usar en la comparacion. Esta lista se pone enmedio de corchetes **[ ]**. Por ejemplo **[abc]** coincide con una letra, la cual puede ser **a**, **b**, o **c**. Es similar a el **?**, pero permite seleccionar con que caracteres coincidira la comparacion. De esta manera **[0-9]** coincide con caracteres **0** hasta el **9**. La unica restriccion en la especificacion de un rango de caracteres es que el primer caracter debe de ser alfabeticamente menor que el ultimo caracter, por lo cual: **[z-f]** no es un rango de caracteres valido.

Mezclando rangos y caracteres en una lista se pueden realizar substituciones mas compilicadas. Por ejemplo **[a-np-z]*** coincide con todos los archivos que comiencen con las letras **a** hasta la **n** o que comiencen con la letra **p** hasta la **z** (en otras palabras, todos los archivos en los cuales la primera letra de su nombre sea cualquier minuscula excepto la letra **o**).

Si el primer caracter de la lista despues de el corchete abierto (**[**) es un signo de admiracion (**!**) el sentido de la comparacion se invierte. Es decir, cualquier caracter, excepto esos que se encuentran en los corchetes, por ejemplo:

[!a-z]

Coincide con cualquier caracter, excepto una letra minuscula, y:

***[!o]**

Es cualquier archivo que no termine con la letra minuscula **o**.

Algunos ejemplos:

Comando	Descripcion
`echo a*`	Despliega los nombre de todos los archivos que comiencen con la letra `a`
`cat *.c`	Despliega el contenido de todos los archivos que terminen con las letras `.c`
`rm *.*`	borra todos los archivos que contengan un punto en cualquier parte de su nombre
`ls x*`	Lista todos los archivos que su nombre comience con `x`
`rm *`	Elimina todos los archivos en el directorio actual
`echo a*b`	Despliega el nombre de todos los archivos que comiencen con la letra `a` y terminen con la letra `b`
`cp ../programas/* .`	Copia todos los archivos del directorio `../programas` a el directorio actual
`ls [a-z]*[!0-9]`	Que hace esto ?

Quiz

Quiz

1 Si se entra a cualquier directorio, digamos el directorio **/tmp** y se teclea el comando:

```
ls -a
```

aparecen 2 archivos, uno llamado punto (.) y otro llamado punto punto (..) cual es la funcion de estos archivos ?

2 Que funcion realiza el siguiente comando:

```
$ ls [a-z][!a-z]*
```

Si en el directorio se encuentran los siguiente archivos:

```
a.out
e-swap.001
iKey.zip
codigo.c
t6
terminal.c
admin
airbone
basura
basura.ordenado
```

3 Si no existe ningun archivo en el directorio y se ejecuta el siguiente comando:

```
$ ls [a-z]*
```

Cual es el resultado del comando ?

4 Cual es el resultado del siguiente comando:

```
$ echo "[a-z]*";
```

5 Cual seria el resultado del comando:

$ cp . . (cp espacio punto espacio punto)

Control de Entrada/Salida

La mayoria de los programas o comandos en los sistemas *nix funcionan de la siguiente manera:

Entrada de datos -> proceso -> salida

Es decir, se introducen datos, se realiza un proceso con esos datos y se obtiene una salida.

Comunmente la forma de introducir datos es el teclado y la salida del comando o programa es desplegada en la pantalla.

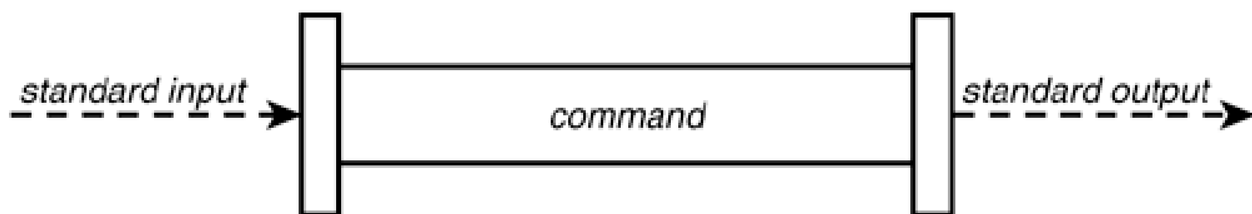
Estos dos elementos, el teclado y la pantalla componen lo que se llama la "entrada estandar" (**stdin**) y la "salida estandar" (**stdout**).

Si queremos desplegar el texto "**hola**", ejecutariamos el siguiente comando:

```
$ echo hola
hola
$
```

La entrada de datos de este comando es el texto **hola** que tecleamos despues del nombre del comando, el proceso es el comando **echo** y la salida la obtenemos despues de teclear **Enter** (que en este caso es la palabra que pedimos que desplegara, **hola**)

Graficamente seria algo como esto



Existen comandos que requieren que el usuario introduzca datos si no se le especifica algun parametro, por ejemplo si tecleamos unicamente el comando cat sin ningun parametro:

```
$ cat
```

El comando no sabe que es lo que va a desplegar debido a que no esta especificado mediante un parametro, por lo cual espera que el usuario introduzca datos para poder procesarlos

```
$ cat
hola
hola
[Control-D]
```

En este caso, el comando **cat** sin ningun parametro espera entrada de datos, al introducir nosotros el texto **hola**, el comando ejecuta el despliegue de ese texto, es por eso que vemos **hola** dos veces, una vez la que nosotros introdujimos y otra la que el comando proceso.

Para salir de este modo especial del comando, tecleamos **Control-D** (**Control-D** envia una señal al proceso indicandole que ha llegao al final del texto, **EOT** o "**End of Text**").

Tambien podemos ver este tipo de comportamiento con el comando **wall**.

El comando **wall** (Write all) envia un mensaje a todos los usuarios que se encuentren en el sistema, si se ejecuta el comando **wall** sin ningun parametro, el comando espera que el usuario introduzca datos para despues comenzar el proceso:

```
$ wall [Enter]
test [Enter]
[Control-D]
Broadcast message from demon (pts/1) Sat Mar 12 22:18:49 2004...

test

$
```

Despues de introducir el mensaje y presionar **Control-D**, el comando **wall** realiza su funcion de enviar el mensaje a todas las terminales.

Aqui podemos comprobar que aunque no parezca, la mayoria de los comandos manejan la entrada de datos (**stdin**) su proceso propio y la salida de datos (**stdout**)

Una de las caracteristicas notables del shell es el control de entrada y salida de los procesos, el shell puede reemplazar la entrada estandar, la salida estandar y el error estandar (todavia no vemos eso, pero mas adelante aparecera nuevamente).

Control de Entrada/Salida

Manipulacion de la salida estandar

Hemos visto que el comando cat sin ningun paramero espera la entrada de datos y repite esta misma entrada de datos en la salida estandar, mediante el shell podemos redirigir la salida estandar para que en lugar de que sea la pantalla sea un archivo.

Esto se logra adicionando al final del comando el caracter **>** (mayor que) y el nombre de archivo al cual queremos que se direccione la salida estandar.

```
$ cat > salida
```

Cuando ejecutamos la linea anterior, el comando cat todavia sigue esperando la entrada estandar para procesarla, es por eso que al dar enter al comando solo se baja la linea en espera de comandos por parte del usuario. Tecleamos algun texto y presionamos Control-D

```
$ cat > salida
hola
como
estas
Control-D

$
```

Anteriormente cuando ejecutabamos el comando **cat** sin ningun parametro pero sin redirigir la salida estandar, lo que tecleabamos aparecia nuevamente en la terminal, en este caso no es asi, debido a que la salida estandar esta ahora redirigida a un archivo llamado salida.

Despues de teclear **Control-D** procedemos a verificar si existe un nuevo archivo llamado salida.

```
$ ls salida
salida

$
```

Y al desplegar el contenido de ese archivo:

```
$ cat salida
hola
como
estas

$
```

Podemos ver lo que inicialmente tecleamos en el comando **cat** (esta tambien es una forma rapida y facil de crear un archivo de texto).

Ahora veamos el comando **sort**, este comando sirve para ordenar la entrada estandar y desplegar la salida estandar ordenada.

Ejecutamos el comando **sort** sin ningun parametro (de esa manera el comando estara esperando datos de la entrada estandar). Y procedemos a introducir datos seguidos de **Control-D**

```
$ sort
a
z
x
8
0
b
9
Control-D
```

Despues de introducir Control-D, el comando sort despliega:

```
0
8
9
a
b
x
z
```

Lo cual es la lista de datos que introdujimos pero ordenada.

Si ejecutamos el mismo comando pero redirigiendo la salida estandar a un archivo:

```
$ sort > lista_ordenada
a
z
x
8
0
b
9
Control-D
```

Despues de introducir **Control-D** el comando sort no despliega la lista ordenada, puesto que su salida de datos esta redirigida a un archivo (en este caso el archivo es lista_ordenada).

Al desplegar el contenido del archivo creado obtenemos la lista ordenada:

```
$ cat lista_ordenada
0
8
9
a
b
x
z
```

Todos los comandos del sistema operativo permiten redirigir su salida a un archivo utilizando el caracter **>** y el nombre de el archivo al cual se dirigira su salida

Por ejemplo, el comando:

```
$ echo hola > archivo_hola
```

Almacenara el texto **hola** en el archivo llamado **archivo_hola**

```
$ cat archivo_hola
hola

$
```

Adicionando la salida

Uno de los problemas de redirigir la salida estandar a un archivo es que cada vez que la salida es enviada al mismo archivo el contenido anterior del archivo se pierde.

Por ejemplo:

```
$ echo hola > mi_archivo
$ echo mundo > mi_archivo
$ cat mi_archivo
mundo

$
```

Para resolver este problema es posible redirigir la salida al archivo adicionando el contenido en lugar de reemplazar lo que ya estaba almacenado ahi. La adicion de la salida se realiza con los caracteres **>>**:

```
$ echo hola > mi_archivo
$ echo mundo >> mi_archivo
$ cat mi_archivo
hola
mundo

$
```

Si el archivo **mi_archivo** no existe, la adicion de contenido (**>>**) creara el archivo, si el archivo ya existe entonces solamente se adiciona el contenido sin eliminar lo anterior. Es decir:

```
$ echo hola > mi_archivo
```

y

```
$ echo hola >> mi_archivo
```

Realizan la misma funcion si el archivo **mi_archivo** no existe antes de ejecutar el comando.

Manipulacion de la entrada estandar

El shell permite manipula la entrada estandar de igual manera que la salida estandar (la entrada estandar normalmente es el teclado)

Para ejemplificar como el shell puede realizar esta accion, regresemos nuevamente al comando **sort**, como habiamos visto, el comando **sort** requiere de la entrada estandar para poder realizar su proceso de ordenamiento.

Lo que haremos es crear un archivo con el comando **cat** (rediriguiendo su salida a un archivo):

```
$ cat > sinorden
z
z
abaco
calculadora
computadora
regla
velocimentro
Control-D

$
```

Hemos creado un archivo llamado **sinorden** que contiene todo lo que tecleamos como entrada estandar a el comando **cat**.

Podemos verificar el contenido de este nuevo archivo con el mismo comando **cat**:

```
$ cat sinorden
z
z
abaco
calculadora
computadora
regla
velocimetro

$
```

Ahora usaremos este archivo como entrada estandar del comando sort:

```
$ sort < sinorden
abaco
calculadora
computadora
regla
velocimetro
z
z

$
```

El comando **sort** lee su entrada estandar (la que requiere para funcionar) del archivo llamado **sinorden** y despliega la lista ordenada.

Podemos incluso combinar la entrada y salida estandar en el comando **sort**:

```
$ sort < sinorden > ordenado
```

Indicandole al comando **sort** que la entrada estandar sea leida desde el archivo llamado **sinorden** y la salida estandar dirigida a el archivo llamado **ordenado**. Al verificar el contenido del archivo **ordenado** con el comando **cat**, podemos ver que **sort** realizo su funcion de ordenar la entrada estandar.

```
$ cat ordenado
abaco
calculadora
computadora
regla
velocimetro
z
z

$
```

Nota: El comando **sort** tiene multiples opciones para ordenar la entrada estandar (ascendente, descendente, sin repeticiones, etc) que mas adelante se veran poco a poco.

Adicion de la entrada (o redireccionamiento inverso)

En el redireccionamiento inverso de la entrada de datos, se puede introducir la entrada estandar de forma automatizada sin tener que teclear la secuencia **Control-D** de fin de texto.

Esto se logra mediante los caracteres **<<** y un identificador de terminacion de entrada, que puede ser cualquier conjunto de caracteres (sin incluir espacio), este conjunto de caracteres tiene que repetirse en el punto que deseamos terminar la entrada de datos.

Anteriormente vimos que con en el comando **sort** se requiere introducir la entrada mediante el teclado o mediante la redireccion de entrada de un archivo:

```
$ sort
zz
aa
bb
Control-D
aa
bb
zz
```

o

```
$ sort < archivo
aa
bb
zz
```

Con redireccionamiento inverso:

```
$ sort << FIN
> zz
> aa
> bb
> FIN
aa
bb
zz
```

En el redireccionamiento inverso, el prompt (o indicador) **>** aparece despues de teclear **Enter**, esto significa que se esta esperando la secuencia de datos que se enviaran como entrada a el comando; esta secuencia termina cuando el identificador indicado al inicio (**FIN**) es encontrado y entonces los datos son enviados al comando como si fueran introducidos mediante el teclado y se enviara al final la combinacion **Control-D**.

Con esto podemos automatizar el comando **wall** que vimos anteriormente:

```
$ wall << FINMENSAJE
> probando mensaje
> FINMENSAJE
Broadcast message from demon Sun Mar 21 02:00:46 2004...

probando mensaje

$
```

Los descriptores de archivos

La entrada estandar (**stdin**) y salida estandar (**stdout**) de cada proceso se identifican internamente mediante descriptores de archivos (**file descriptors**).

Cuando un proceso abre un archivo, el sistema operativo le asigna un numero secuencial consecutivo. Esto es con el fin de poder identificar y diferenciar los archivos abiertos, este numero consecutivo secuencial es llamado **descriptor de archivo** o **file descriptor**.

Los descriptores de archivos para el proceso comienzan con el numero **3**, debido a que los numero **0**, **1** y **2** ya estan utilizados por el sistema.

Para cualquier proceso que se ejecute en el sistema, la entrada estandar (entrada de instrucciones del teclado) es el descriptor de archivo numero cero (**0**) y la salida estandar es el descriptor de archivo numero uno (**1**).

Existe adicionalmente otro descriptor de archivo del que no hemos comentado, este descriptor de archivo es el numero **2** y es llamado error estandar (**standard error** o **stderr**)

El estandar error o error estandar o stderr es metodo mediante el cual los comandos y programas del sistema despliegan sus errores. Para ilustrar esto, intencionalmente ejecutemos un comando con un parametro erroneo:

```
$ cat archivo_que_no_existe
cat: archivo_que_no_existe: No such file or directory

$
```

El comando despliega un error debido a que el archivo que tratamos de mostrar no existe.

Para cualquier proceso, la entrada estandar es el teclado y la salida estandar es la pantalla; la salida para el error estandar tambien es la pantalla; es por eso que aveces es poco dificil distinguir la salida estandar de el error estandar ya que la salida de ambos es la misma (la pantalla).

Debido a que el error estandar (o salida de errores del comando) tiene un descriptor de archivo diferente a la salida estandar (o salida de resultados del comando), es posible rediriguir a un archivo el error estandar de igual manera que hemos hecho con la salida estandar.

Para redirigir el error estandar se utiliza el mismo caracter **>** que utilizamos para dirigir la salida estandar anteriormente, el unico cambio es que tenemos que especificarle a el shell que deseamos redirigir un descriptor de archivo especifico, el descriptor numero 2, el cual esta asociado con la salida de errores del comando:

```
$ cat archivo_que_no_existe 2> error
$
```

El mensaje de error que el proceso **cat** desplegaba ya no aparece en la pantalla; esta redirigido a un archivo, el nombre del archivo es **error**.

Podemos ahora ver el contenido del archivo llamado **error** y verificar que el mensaje de error del comando **cat** esta almacenado ahi.

```
$ cat error
cat: archivo_que_no_existe: No such file or directory

$
```

De igula manera que se puede adicionar la salida estandar tambien se puede adicionar la salida del error estandar:

```
$ cat otro_que_no_existe 2>> error
$ cat error
cat: archivo_que_no_existe: No such file or directory
cat: otro_que_no_existe: No such file or directory

$
```

Control de Entrada/Salida

Pipes (Entubamiento de comandos)

La mayoria de los comandos de Unix que estan diseñados para trabajar con archivos pueden tambien obtener los parametros que requieren para su funcionamiento de la entrada estandar. Esto permite usar un programa para filtrar la salida de otro. Esta es una de las operaciones mas comunes en la programacion shell: hacer que un programa manipule la salida de otro.

Se puede direccionar la salida de un comando como entrada de otro utilizando un pipe (**|**) el cual permite "conectar" 2 o mas comandos.

La sinaxis general para "conectar" 2 o mas comandos es:

```
comando1 | comando2 | ...
```

El caracter pipe (**|**) conecta la salida estandar del comando1 a la entrada estandar de el comando2 y asi sucesivamente. Los comandos pueden ser tan simples o complejos como sea requerido

Para poner un ejemplo, si quisieramos saber cuantos usuarios se encuentran en este momento en el sistema, una forma facil seria la de contar el numero de lineas (con el comando **wc**) que el comando **who** despliega (debido a que existe una linea por cada usuario que se encuentra en linea en el sistema), sin usar pipes, tendriamos que hacer algo asi:

```
$ who > usuarios
$ wc -l < usuarios
    5

$
```

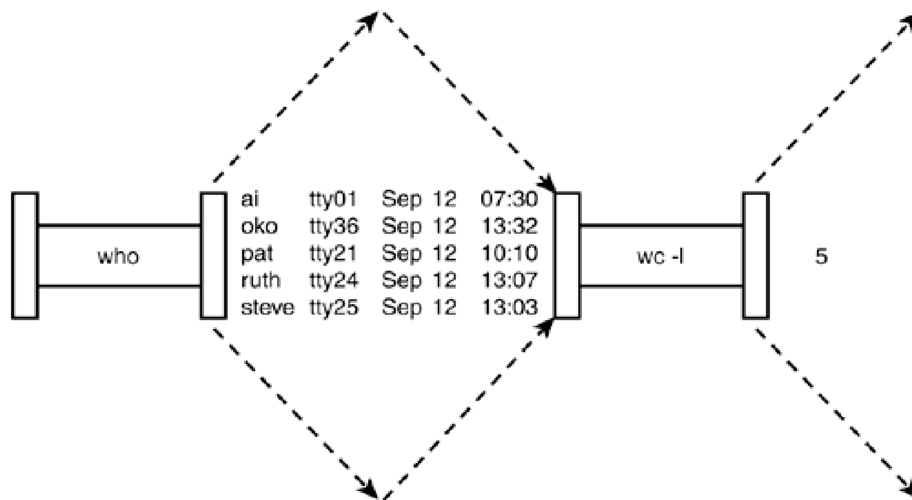
Redirigimos la salida del comando **who** a el archivo llamado **usuarios**, despues redirigimos este archivo como entrada del comando **wc** (con el parametro **-l** para poder contar las lineas), el resultado es el numero de lineas que se encuentran en el archivo **usuarios** y por lo tanto el numero de usuarios en el sistema.

Otra manera de obtener este mismo resultado sin involucrar la creacion de un archivo intermedio, es el uso de pipes. Conectar la salida de datos del comando **who** con la entrada de datos del comando **wc** de la siguiente manera:

```
$ who | wc -l
    5

$
```

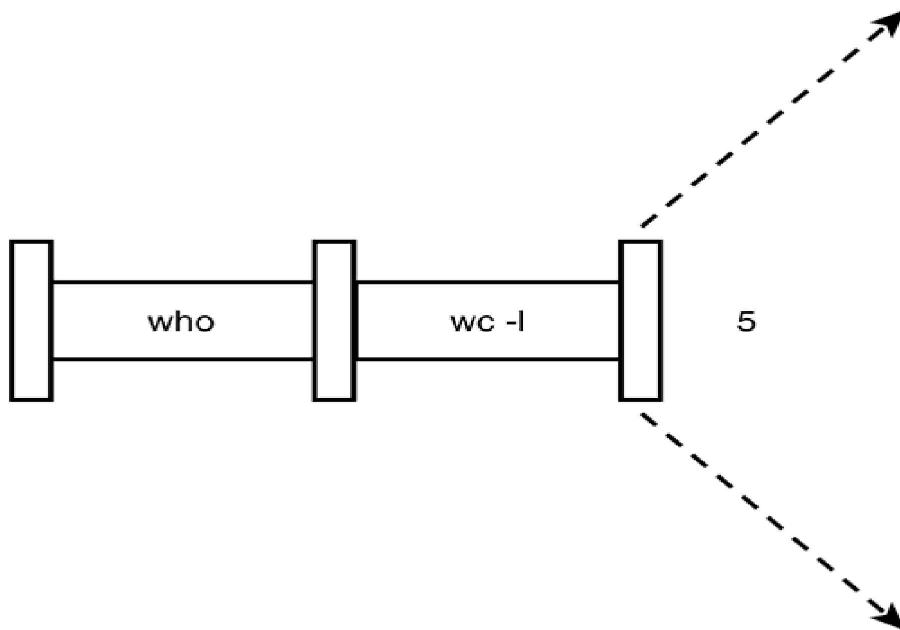
El efecto que el pipe crea en estos dos comandos podria ejemplificarse de la siguiente forma:



La salida del comando **who** es enviada como entrada a el comando **wc**. Es por eso que el unico dato desplegado en la pantalla es el numero **5**, resultado de la salida del comando **wc**; la salida del comando **who** no aparece en la pantalla debido a que es conectada a el comando **wc**. Creando un solo comando compuesto

Nota: **tee** puede usarse para almacenar la entrada estandar y redirigirla.

Este comando unificado podira describirse graficamente de la siguiente manera:



Un pipe puede ser creado entre cualquier par de programas, siempre y cuando el primer programa envie su salida a la salida estandar (**stdout**) y el segundo programa lea su entrada de datos de la entrada estandar (**stdin**).

Como otro ejemplo, supongagmos que se desea saber el numero de archivos existententes en el directorio. Sabiendo que el comando **ls** despliega una linea por cada archivo existente podriamos utilizar la misma tecnica que en el ejemplo anterior:

```
$ ls | wc -l
    10

$
```

La salida indica que el directorio actual contiene 10 archivos.

Tambien es posible crear pipes consistiendo de mas de 2 comandos o programa, con la salida de uno alimentando la entrada del siguiente comando.

Filtros

El termino filtro es comunmente utilizado en la terminologia Unix para referirse a cualquier programa que pueda tomar su entrada de datos de la entrada estandar (**stdin**), realizar alguna operacion basado en esa entrada de datos, y enviar el resultado de la operacion a la salida estandar (**stdout**). Mas estrictamente, un filtro es un programa que puede ser utilizado utilizado enmedio de otros dos programas en un pipe, asi es que en los ejemplos anteriores, **wc** es considerado un filtro. Otros ejemplos de filtro serian los comandos: **sort**, **cat** y **cut**.

cut es un programa que permite dividir la entrada de datos que se le asigna y separarla en base a caracteristicas que se le definan (un delimitador).

cut se basa en la identificacion de un caracter como delimitador de campos, una vez delimitados los campos, **cut** puede desplegar el campo deseado. por ejemplo en la siguiente cadena:

dato1:dato2:dato3:dato4

El caracter que sirve para separar los datos (para indicar donde termina un dato y donde empieza otro) son los dos puntos verticales (**:**), se le podria indicar a el comando **cut** que el delimitador de los campo es los dos puntos (**:**) mediante el parametro **-d** (**delimiter**) seguido de el caracter delimitador, una vez delimitados los campos, se puede extraer el campo deseado con el parametro **-f** (**field**) del comando **cut**.

Por ejemplo, para traer el segundo campo de la cadena anteriormente descrita (**dato2**) se utilizaria el siguiente comando:

```
cut -d: -f2
```

Esto le indica a el comando **cut** que el delimitador seran los dos puntos (**-d:**) y que se desea que despliegue el campo numero 2 (-**f2**).

El comando **cut** lee los datos a separar, de la entrada estandar, es por eso que si ejecutamos el comando:

```
$ cut -d: -f2
```

cut espera que se introduzcan datos para poder realizar la operacion solicitada, si introducimos

abc:def

cut desplegara el segundo campo (**def**) de acuerdo a los parametros especificados anteriormente.

```
$ cut -d: -f2
abc:def
def
Control-D

$
```

Ya que el comando **cut** lee los datos a procesar de la entrada estandar es posible incluirlo con otro comando en un pipe:

```
$ echo "dato1:dato2:dato3" | cut -d: -f2
dato2
```

Analizando el archivo **/etc/password** vemos los siguientes datos:

```
$ cat /etc/passwd
root:x:0:0:root:/root:/bin/bash
bin:x:1:1:bin:/bin:
daemon:x:2:2:daemon:/sbin:
adm:x:3:4:adm:/var/adm:
lp:x:4:7:lp:/var/spool/lpd:
shutdown:x:6:0:shutdown:/sbin:/sbin/shutdown
halt:x:7:0:halt:/sbin:/sbin/halt
mail:x:8:12:mail:/var/spool/mail:
gopher:x:13:30:gopher:/usr/lib/gopher-data:
ftp:x:14:50:FTP User:/var/ftp:
$
```

El separador de campos es **:**, existen 7 campos en cada linea los cuales son:

- El nombre del usuario
- El password del usuario (el cual no se almacena en este archivo por cuestiones de seguridad)
- El identificador numerico del usuario (**user id** o **uid**)
- El grupo de usuarios al que pertenece el usuario (**group id** o **gid**)
- El nombre completo del usuario
- Su directorio de casa (**home directory**)
- El programa a ejecutar cuando el usuario entre al sistema

Quiz 2

1 Que cual sera la salida de la siguiente linea de comando:

```
$ echo "datos" | echo | echo
```

y porque ?

2 Cual es la diferencia entre los siguientes comandos:

```
$ echo "[a-n]*" | cat
```

y

```
$ echo [a-n]* | cat
```

3 Tomando en cuenta los datos de el archivo **/etc/password** mostrados en la pagina anterior con que comando podrian desplegarse unicamente el nombre de los usuarios ?

Y que haria el siguiente comando:

```
$ cat /etc/hosts | cut -dp -f2
```

4 Cual es el resultado del siguiente comando:

```
$ echo "hola" | cat | cat | cat
```

5 Tomando en cuenta que en el directorio existen los siguiente archivos:

```
programa1.c
programa2.c
applet.java
applet2.java
noext
```

Que funcion realiza el siguiente comando:

```
$ ls | cat | cut -d. -f2
```

6 Que hace la siguiente linea de comando:

```
$ date | cut -f100 >> fecha
```

Variables

Asignacion de variables

Las variables son "palabras" con contienen un valor. El shell permite crear, asignar y eliminar variables.

El nombre de una varaible puede contener solo letras (de **a** a **z** o de **A** a **Z**), numeros (**0** a **9**) o el caracter de guion bajo (**_**). Adicionalmente el nombre de una variable solo puede comenzar con una letra o un guion bajo.

Estos son algunos ejemplos de nombres de varaibles validas:

_TOTAL
NUMERO_TOTAL
archivo1
_datos
nombre_de_archivo

Pero:

2resultado

no es nombre de variable valido, se puede agregar un guion bajo al principio para convertirla en valida:

_2resultado

Nombres de variables tales como **1**, **2** o **11**. Es decir que comienzan con un numero, estan reservadas para ser utilizada por el shell (**file descriptors**), se puede utilizar el valor almacenado en esas varaibles pero generalmente tu no puedes poner el valor de esas variables.

La razon por la que no se puede utilizar caracteres como **!**, ***** o **-** es que estos caracteres tienen un significado especial para el shell (substitucion de nombres de archivos).

A este tipo de variables simples se les llamada tecnicamente "escalares" (**scalar**).

Variables

Asignando valores a las variables

El shell permite asignar practicamente cualquier valor a las variables simplemente especificando el nombre de la variable y el valor que queremos almacenar en esa variable. Por ejemplo:

```
$ FRUTA=manzana
$ FRUTA=kiwi
$ Total=358
```

No hay que declarar el tipo de variable (caracter o numerica) pero hay que tener cuidado al utilizar valores que contienen espacios, por ejemplo:

```
FRUTA=manzana naranja pera
```

Generara el siguiente error:

```
$ FRUTA=manzana naranja pera
bash: naranja: command not found
```

Para poder utilizar espacios en los valores de las variables, se requiere que el valor se encierre entre comillas (").

```
$ FRUTA="manzana naranja pera"
```

Accesando el valor de las variables

Para obtener el valor almacenado en una variable se incluye el signo de pesos (**$**) antes de el nombre de la variable, por ejemplo:

```
$ FRUTA=pera
$ echo $FRUTA
pera
$ total=385
$ echo $total
385
$
```

Hay que tomar en cuenta que si no se utiliza el signo de pesos (**$**) antes del nombre de la variable, unicamente se despliega el nombre de la variable, por ejemplo:

```
$ FRUTA=platano
$ echo FRUTA
FRUTA
$
```

El signo de pesos (**$**) es unicamente utilizado para acceder a el valor de las variables, pero no para definirlo, por ejemplo, la siguiente asignacion:

```
$ $FRUTA=manzana
```

Generara el siguiente error

```
bash: $: command not found
```

Se puede tambien asingar el valor de otras variables a una variable por ejemplo:

```
$ X=hola
$ Y="$X como estas"
$ echo $Y
hola como estas

$
```

Nota: se puede escapar el caracter **$** con el backshash (****)

Aritmetica integrada

El shell provee un mecanismo para realizar aritmetica de enteros basica llamado "expansion artimetica". Este tipo de expansion no esta disponible en shells mas antiguos (como Bourne shell, donde se tiene que utilizar el comando **expr** que veremos mas adelante).

La expansion aritmetica esta basada en un estandar **POSIX**, asi que es posible encontrar esta caracteristica en la mayoria de shells modernos.

El formato para la expansion aritmetica es:

```
$((expresion))
```

En donde expresion es cualquier expresion matematica que puede usar variables de shell (que contengan valores numericos), operadores y/o valores numericos especificos.

El resultado de el calculo de la expresion es regresado en la linea de comando, por ejemplo:

```
echo $((i+1))
```

Adiciona uno a el valor de la variable **i** y despliega el resultado, hay que notar que la variable i no tiene que ser precedida por el signo de pesos (**$**). Esto es porque el shell sabe que las unicas cuestiones validas en una expansion aritmetica son valores, variables y operadores (**+ - * /**), al evaluar el caracter i y definir que no es un valor numerico (**0** a **9**) ni un operador matematico, entonces seguramente es una variable.

Si la variable no ha sido definida o su valor es nulo (contiene una cadena nula, espacios o valores alfanumericos) se define que el valor de la variable es cero. Por ejemplo, aun cuando no hemos definido la variable **Z** ni asignado ningun valor a la misma, aun se puede utilizar en una expansion aritmetica:

```
$echo $((Z+10))
10
$
```

o en la siguiente:

```
$ echo $((Z=Z+1))
1
$
```

Ahora **Z** contiene el valor **1**.

La asignacion de valor (**=**) es un operador aritmetico valido y el valor de el calculo es asignado a la variable.

Se permite el uso de parentesis adicionales para forzar agrupacion de operaciones, por ejemplo:

```
echo $((i=(i+10) * j))
```

Si se desea realizar una asignacion sin el comando echo (el cual despliega el resultado de la operacion en la terminal), se mueve la asignacion de datos (el signo de igual) antes de la expansion aritmetica.

```
$ total=$((1 * 10))
$ echo $total
10
$
```

Finalmente, para hacer una comparacion de valores dentro de la asingacion, digamos, ver si es mayor que 0 o menor o igual que 100, podemos escribir:

```
$ resultado=$(( i >0 && i <=100 ))
```

Lo cual asigna el valor 1 a la variable resultado si la expresion es cierta y 0 si es falsa.

Nota: && es AND logico y || or logico:

Asignando la salida estandar a variables

Una caracteristica muy util de la asignacion de variables es poder asignar resultados de comandos (o la salida estandar de un comando) a variables utilizando el caracter ` para encerrar el comando.

Supongamos que deseamos una variable que contenga la fecha del sistema, podriamos hacer lo siguiente:

```
$ FECHA=`date`
```

Notar el caracter ` al comenzar y terminar el comando.

De esta manera la variable **FECHA** contendra el resultado de la ejecucion del comando **date**.

```
$ echo $FECHA
Sun Mar 21 04:41:57 CST 2004
```

O por ejemplo, para asignar el numero de usuarios que se encuentran en el sistema:

```
$ NUMUSR=`who | wc -l`
$ echo $NUMUSR
5
```

O el numero de procesos ejecutandose en el sistema:

```
$ PROCNUM=`ps -ef | wc -l`
$ echo $PROCNUM
72
```

O almacenar el contenido de un archivo en una variable

```
$ HFILE=`cat /etc/hosts`
```

Este tipo de substitucion es ampliamente utilizado en la programacion shell, debido a que permite almacenar en una variable la salida de un programa o incluso de un conjunto de porgramas (unidos mediante uno o varios pipes, como en el ejemplo de: **ps-ef | wc -l**).

Arrays

Un array es un conjunto de valores asingados a un solo nombre de variable, para accesar alguno de los valores, se utiliza un numero que sirve como indice para los datos.

Digamos que deseamos almacenar todos los nombres de frutas en una sola variable a la que llamaremos FRUTA.

Si ejecutamos:

```
$ FRUTA=manzana
$ FRUTA=pera
$ FRUTA=naranja
```

El problema es que la variable **FRUTA** contendra unicamente el ultimo valor asignado (**naranja** en este caso) y no todos los valores.

Una variable de tipo array formaliza este tipo de agrupacion de valores usando un nombre de variable en conjunto con un numero para accesar los valores, este numero es llamado indice.

El metodo mas simple para crear un array es asignar un valor a uno de sus indices, esto es expresado de la siguiente manera:

```
nombre[indice]=valor
```

En donde **nombre** es el nombre del array, **index** es el indice a la celda que se desea asignar y **valor** es el valor a asignar.

Por ejemplo:

```
$ FRUTA[0]=manzana
$ FRUTA[1]=pera
$ FRUTA[2]=naranja
```

En bash, el valor de indice comienza con cero cuando se utilizan valores numericos

No debe de existir un espacio entre el corchete inicial ([) y el valor del indice, debido a que:

```
$ FRUTA[ 1]=manzana
```

generara el siguiente error:

```
bash: FRUTA[: command not found
```

Porque el espacio que se encuentra entre el nombre de variable y el indice le dice a el shell que **FRUTA[** es un comando y que **1]=manzana** es un parametro para ese comando, por lo cual, el shell trata de encontrar un programa llamado **FRUTA[.**

El valor de indice no necesariamente tiene que ser un valor numerico, tambien se pueden asignar valores de indice que representen datos alfanumericos, por ejemplo:

```
$ FRUTA[primera]=manzana
$ FRUTA[segunda]=pera
$ FRUTA[tercera]=naranja
```

Tambien es posible asignar los valores de un array especificando todos lo valores posibles entre parentesis (**()**), por ejemplo:

```
$ FRUTA=(manzana pera naranja)
```

Es equivalente a:

```
$ FRUTA[0]=manzana
$ FRUTA[1]=pera
$ FRUTA[2]=naranja
```

Si desamos asignar algun valor que contenga espacios a un elemento del array, tenemos que encerrarlo entre comillas de igual manera que si fuera una variable comun y corriente:

```
$ FRUTA[100]=uvas verdes
bash: verdes: command not found
$ FRUTA[100]="uvas verdes"
```

Asi, podriamos:

```
$ FRUTA=(manzana pera naranja "uvas verdes" "uvas rojas")
```

Accesando el valor de variables de Array

Despues de que se ha asignado un valor a una variable de array, se puede accesar de la siguiente manera:

```
${nombre[indice]}
```

Donde nombre es el nombre de el array, e indice es el indice del elemento que deseamos. Por ejemplo:

```
$ FRUTA[0]=manzana
$ FRUTA[1]=platano
$ echo ${FRUTA[0]}
manzana
$ echo ${FRUTA[1]}
platano
$ FRUTA[rara]=kiwi
$ echo ${FRUTA[rara]}
kiwi
$
```

Se pueden accesar todos los elementos de un array utilizando como indice los caracteres **@** o ***** de la siguiente manera:

```
$ computadoras=(apple ibm dell acer toshiba "bull systems")
$ echo ${computadoras[@]}
apple ibm dell acer toshiba bull systems
$ echo ${computadoras[*]}
apple ibm dell acer toshiba bull systems
$
```

Variables de solo lectura

Una vez que una variable ha sido definida, es posible indicarle al shell que deseamos que sea de solo lectura, es decir que el valor que tiene asignado no se pueda modificar, esto es utilizado algunas veces en programas de shell para asegurarse de que valores importantes que esas variables contienen no sean modificados durante la ejecucion.

```
$ C=123
$ echo $C
123

$ readonly C
$ echo $C
123
$ C=200
bash: C: readonly variable

$
```

Aun cuando la variable es de solo lectura el comando **echo** todavia puede obtener su valor, pero cuando trata de cambiarse el valor a cualquier otro el shell no lo permite.

Tambien es posible definir a arrays como de solo lectura.

```
$ A[0]=hola
$ A[1]=200
$ A[2]=foo
$ readonly A
$ A[1]=300
bash: A: readonly variable
```

Para eliminar una variable o array, se puede utilizar el comando unset:

```
$ C=100
$ echo $C
100
$ unset C
$ echo $C

$
```

Variables

Variables de ambiente

Cuando el shell esta ejecutandose, tres tipos de variables estan presentes:

- Variables locales
- Variables de ambiente
- Variables del shell

Una variable local esta presente en la instancia actual del shell. No esta disponible a programas que han sido invocados por el shell. Todas las variables que hemos creado en las paginas anteriores son variables locales

Una variable de ambiente es una variable que esta disponible a cualquier proceso invocado por el shell, algunos programas requieren variables de ambiente para funcionar correctamente.

Una variable de shell es una variable que esta definida por el shell y es requerida por el mismo para funcionar correctemamante. Algunas de estas variables son variables de ambiente y algunas otras son locales.

Para desplegar todas las variables (y sus valores) que el shell controla podemos utilizar el comando **`set`**:

```
$ set
```

(lo cual regresara un largo listado de variables de ambiente y locales)

Tambien es posible utilizar set para definir una variable:

```
$ set FRUTA=manzana
```

Algunos programas de shell definen sus variables con el comando **`set`**, algunos sin utilizar el comando **`set`** (directamente especificando el nombre y el valor), no existe diferencia alguna en usar **`set`** o no usarlo para definir variables.

Exportando variables a el ambiente

Es posible convertir una variable local a una variable de ambiente con el comando **export**.

Una variable de ambiente puede ser accesada por cualquier programa que el shell ejecute; esta es una de las razones principales para exportar variables.

El formato de el comando **export** es:

```
export nombre_de_variable
```

Tambien es posible utilizar el comando **export** para definir la variable, su valor y al mismo tiempo convertirla en una variable de ambiente:

```
$ export C=359
```

Se puede utilizar el comando **export** para convertir mas de una variable local a variable de ambiente especificando todas las variables en una lista:

```
$ A=9
$ B=3
$ C=300
$ export A B C
```

Utilizando asignacion:

```
$ export A=9 B=3 C=300
```

Mezclando referencia a la variable y asignacion:

```
$ A=9
$ export C=4 D=93 A
```

Variables

Variables del shell

Las variables que hemos examinado hasta este punto son variables de usuario. Una variable de usuario es aquella que el usuario puede definir y eliminar de forma manual.

Las variables del shell, son variables que el shell define durante su inicializacion y las utiliza internamente para realizar ciertas tareas.

La siguiente tabla provee una lista parcial de las variables del shell **`bash`** y su funcion:

Variable	Descripcion
`PWD`	Indica el directorio actual de trabajo
`UID`	Contiene el numero de usuario para el sistema
`SHLVL`	Se incrementa cada vez que una instancia de **`bash`** es ejecutada. Esta variable es util para determinar si el comando **`exit`** termina la sesion.
`RANDOM`	Genera un numero aleatorio entre `0` y `32767`
`SECONDS`	Contiene el numero de segundos que el shell lleva ejecuntadose. Si se le asigna un valor numerico entonces llevara el conteo en segundos apartir de el valor asignado
`IFS`	Indica el separador interno de de campos, es utilizado para determinar donde comienzan y terminan los campos de una cadena, veremos este mas adelante en **`awk`**.
`PATH`	Indica las rutas de busqueda para los comandos. Es una lista de directorios separada por dos puntos (**`:`**).
`HOME`	Contiene el directorio de casa de el usuario actual, es el argumento de default para el comando **`cd`**
`HISTFILE`	Contiene la ruta para el archivo historico de comandos ejecutados

La variable de resultados

Una variable muy util en la programacion shell es la variable de resultados, esta es una variable de ambiente que contiene el resultado del ultimo comando.

Normalmente cuando se crea algun programa para ambiente *nix este se codifica en lenguaje C, si se utiliza la representacion formal de programacion (es decir, se programa como debe de hacerse) la rutina principal debe de regresar un valor entero, segun el prototipo especificado como estandar:

```
int main(int argc, char *argv[])
```

El valor entero que la rutina main regresa es almacenado por el shell en la variable **?** (signo de interrogacion).

Los programadores aprovechan el tener que regresar un valor numerico en su programa para informar al shell si el comando ejecutado fue exitoso o surgio algun error. Es de hecho el regresar el estatus general del programa.

En el manual de la mayoria de los comandos del sistema especifican el valor de regreso de el comando, este valor de regreso es el valor numerico almacenado en la variable **?**, mediante el cual, el programa en shell podra saber si el comando se ejecuto con exito o no.

Por ejemplo:

Viendo la pagina de manual del comando **cat** ($ **man cat**) encontramos lo siguiente:

```
The cat utility exits 0 on success, and >0 if an error occurs.
```

Es decir, que el comando **cat** regresara el valor **0** si el comando se ejecuto de forma correcta y un valor mayor a **0** si existio algun error en su ejecucion. Este valor es capturado por el shell y almacenado en la variable de ambiente **?**.

De la siguiente manera, esta linea de comando que sabemos de antemano que es erronea:

```
$ cat adsfasfdsaf
cat: adsfasfdsaf: no such file or directory
$ echo $?
1
$
```

Almacena en la variable **$?** el valor **1**.

Y la siguiente, la cual hace que el comando **cat** funcione correctamente:

```
$ cat /dev/null
$ echo $?
0
$
```

Almacena en la variable **$?** el valor **0**.

Cuando se utilizan **pipes** para conectar la salida y entrada de comandos, el ultimo comando en la lista de **pipe** es quien asigna el valor a la variable **?**.

```
$ echo "adfadfasfd:asdfasdfasdf" | cut -dadfasdf
usage:  cut -c list [file1 ...]
        cut -f list [-s] [-d delim] [file ...]
$ echo $?
1
$
```

En este caso, el comando **echo** funciona correctamente y envia su salida al comando **cut**, el cual tiene un error de sintaxis, haciendo que la variable **?** regrese el valor **1** tal y como lo dice la pagina de manual de dicho comando.

```
$ cat adfadsf | cut -d: -f2
cat: adfadsf: No such file or directory
$ echo $?
0
$
```

El comando **cat** es el que genera el error, al hacer esto no hay salida estandar del comando y no envia nada como entrada de datos al comando **cut** (bueno, realmente si envia algo, envia unicamente la señal de fin de texto, **EOT** o **Control-D**).

El comando **cut** actua como si se hubiera sido invocado de la siguiente manera:

```
$ cut -d: -f2 << EOT
> EOT
$
```

El resultado numerico de la variable **?** esta asignado por el comando cut, no por el comando cat (el comando cat asignaria la variable ? con un valor 1 al haber tenido una ejecucion erronea).

Es decir: la variable **?** contiene el resultado numerico del ultimo comando.

Es normalmente utilizado para poder realizar cuestiones condicionales, por ejemplo:

Segun el manual del comando grep, dice:

Normally, exit status is 0 if matches were found, and 1 if no matches were found

Si deseamos saber si existe algun usuario en el archivo /etc/passwd, podriamos

```
usuario="juan"
cat /etc/passwd | grep $usuario > /dev/null
if [ "$?" -eq "0"]; then
    echo "el usuario $usuario, si existe en /etc/passwd"
else
    echo "el usuario $usuario, no existe en /etc/passwd"
fi
```

Utilizamos el direccionamiento a **/dev/null** porque si no lo hacemos el comando **grep** desplegara la linea completa del archivo **/etc/passwd** en caso de que si se encontrara el usuario ahi, de esta manera evaluamos el codigo de salida del comando **grep**, si el comando grep encontro a el usuario (salida 0) o si no lo encontro (salida 1). Las comillas adicionadas a la variable **$?** son por seguridad, y mas adelante hablaremos de eso.

Variables

Consturccion de variables

Supongamos que se tiene el nombre de un archivo almacenado en la variable **ARCH**. Si se quisiera renombrar ese archivo de manera que el nuevo nombre tuviera una **X** al final del archivo, lo primero que tratariamos de hacer es:

```
$ mv $ARCH $ARCHX
```

Cuando el shell lee la linea de comando, substituye el valor de la variable **$ARCH** por el nombre de archivo almacenado en la misma y tambien el valor de la variable **$ARCHX** (la cual no existe), el shell piensa que **$ARCHX** es una variable. Para evitar este problema, se puede delimitar el final de el nombre de una variable encerrandola entre llaves (sin encerrar el signo de pesos para referenciarla **$**). De la siguiente manera:

```
${ARCH}X
```

Esto elimina la ambiguedad y el comando **mv** puede funcionar como deseamos

```
$ mv $ARCH ${ARCH}X
```

Una encapsulacion como esta (el uso de las llaves) es necesario solamente si el caracter despues de el nombre de la variable es un caracter alfanumerico o un guion bajo, en el siguiente ejemplo, esta encapsulacion no es necesaria:

```
$ C=archivo
$ echo $C-nuevo
archivo-nuevo

$
```

Control de procesos

Control de procesos

Cuando se invoca algun comando en Unix, el sistema crea o inicia un nuevo proceso. En las paginas anteriores, cuando hemos utilizado el comando ls para listar el contenido de los directorios, hemos generado un proceso (el proceso creado por la ejecucion del comando ls)

El sistema operativo lleva control de los procesos a traves de un identificador numerico de cinco digitos para cada proceso; este identificador es conocido como process id (**`pid`**).

Cada proceso en el sistema tiene un **`pid`** unico. Eventualmente los pid's se repiten debido a que todas las combinaciones numericas disponibles estan utilizadas y el identificador numerico comienza nuevamente a contar desde el principio. No pueden existir dos procesos con el mismo pid en el sistema. El contador de numeros de procesos vuelve a el inicio despues de llegar a el maximo de un entero de 16 bits (32,767).

Cuando se inicia un proceso, existen dos maneras de ejecutarlo:

- En el frente (**`Foreground`**)
- En la parte de atras (**`Background`**)

La diferencia entre estos dos modos de ejecucion es como el proceso interactua con el usuario y la terminal.

Procesos en Foreground

Por default cualquier proceso que se inicie se ejecuta en Foreground. Obtiene su entrada del teclado y envia su salida a la pantalla. Se puede redireccionar la entrada y salida del proceso como hemos visto anteriormente, pero por default la entrada y salida estan direccionadas a la terminal.

Se puede comprobar lo anterior con el comando **`ls`**. Si se desea listar todos los archivos que comiencen con la palabra programa en el directorio se usaria el siguiente comando:

```
$ ls programa*
programa1.c
programa2.python
programa3.pl

$
```

El proceso se ejecuta en **`Foreground`**, la salida es dirigida a la pantalla y si el comando **`ls`** requiriera alguna introduccion adicional de datos (que no es el hecho), esperaria la entrada de datos desde el teclado.

Control de procesos

Ahora veamos el comando **sleep**, este es un comando sencillo, recibe un parametro numerico y espera (duerme) hasta que transcurra el numero de segundos que se le indicaron.

Por lo anterior:

$ sleep 10

Espera **10** segundos

$ sleep 3

Espera **3** segundos

Si ejecutamos

$ sleep 150

El comando **sleep** estara esperando **150** segundos antes de regresar el control al shell (antes de que termine el proceso).

Mientras este periodo de tiempo pasa (la ejecucion del comando), no se puede ejecutar ningun otro comando. Se pueden teclear comandos, pero no aparece el prompt del shell (**$**) y nada parece funcionar hasta que el comando que esta ejecutandose se complete, esto es porque Unix envia la entrada de teclado a un buffer.

El shell provee la facilidad de no tener que esperar que un proceso se complete antes de comenzar otro, esto es llamado, ejecucion en **background**.

Adicionalmente el shell tiene la capacidad de suspender procesos (hacer una pausa en su operacion) que se encuentren en **foreground** o en **background** e incluso puede mover procesos entre **foreground** y **background**.

Control de procesos

Procesos en background

Un proceso en **background** se ejecuta sin estar conectado a la terminal, si el proceso en **background** requiere entrada de datos (del teclado), se detendra.

La ventaja de ejecutar un proceso en **background** es que se pueden ejecutar otros comandos; no se tiene que esperar a que se termine un proceso especifico antes de lanzar otro (como en el caso anterior, que teniamos que esperar que terminara el comando **sleep** para poder ejecutar otro comando)

Para enviar un proceso a **background** se utiliza el caracter **&**, este caracter se adicional al final de la linea de comando que normalmente ejecutariamos, por ejemplo:

Para hacer que el proceso **sleep** espere por 3 segundos, normalmente ejecutariamos:

```
$ sleep 3
```

Para hacer que ese mismo proceso realice su funcion pero en **background** ejecutamos:

```
$ sleep 3 &
```

Cuando lanzamos un proceso con el caracter **&** (indicandole que deseamos que se ejecute en **background**) el shell le pide a el sistema que genere un nuevo proceso, el sistema asigna un numero secuencial a este nuevo proceso y le reporta al shell cual es ese numero, entonces el shell reporta a el usuario cual es el numero de el proceso que el acaba de lanzar.

Tal como en el proceso de **sleep** que acabamos de lanzar:

```
$ sleep 3 &
[1] 6278

$
```

El shell indica que el numero de proceso que se le ha asignado a este nuevo proceso es el **6278** (el numero de proceso seguramente sera diferente si pruebas esto en tu propia terminal), y que es el primer trabajo que tengo corriendo en **background**.

Inmediatamente regresa el prompt del shell, permitiendonos ejecutar algun otro proceso o comando

Cuando el proceso en **background** ha terminado su ejecucion (si es que no requiere entrada de teclado), el shell informa a el usuario que el proceso ha sido completado con un mensaje similar a este:

```
[1]   Done                    sleep 3
```

Podriamos ahora probar el mismo comando **sleep** pero con un poco mas de tiempo:

```
$ sleep 1500 &
[1] 6283
```

El nuevo proceso que enviamos a **background** ahora tiene el numero de proceso **6283** y es unico trabajo que tenemos en background (el **1** entre corchetes), si enviamos otro:

```
$ sleep 300 &
[2] 6284
```

Notamos lo siguiente:

1.- Este es el segundo trabajo que tenemos ejecutandose en background (**[2]**)
2.- Efectivamente los numeros de proceso son secuenciales (**6283, 6284**)

Si verificamos mendiante el comando **ps** cuantos procesos estamos ejecutando

```
$ ps
  PID  TT  STAT      TIME COMMAND
 6239 std  Ss     0:00.10 -bash
 6283 std  S      0:00.00 sleep 1500
 6284 std  S      0:00.01 sleep 300
```

Podemos ver que ahi se encuentran los 2 procesos que enviamos a **background**.

Si se envia un proceso que requiere entrada de datos a **background** (tal como el comando **cat** o el comando **wall**) el proceso se detiene, debido a que no puede continuar con su funcion normal sin entrada de datos, por ejemplo:

```
$ cat &
[4] 6293
```

En donde el shell nos indica que ha enviado a **background** el proceso **cat**, este es el tercer trabajo que tenemos en **background** (puede variar el numero consecutivo de proceso en background si tu lo pruebas en tu terminal) y que es el numero de proceso **6293** (el cual tambien puede variar en tu terminal). Despues de enviar el proceso a background y debido a que el proceso cat requiere entrada de datos, el shell detiene el proceso y envia el siguiente mensaje a la terminal:

```
[3]+  Stopped                    cat
```

El tercer proceso que enviamos a background se ha detenido puesto que no puede continuar sin entrada de datos.

Para poder ejecutar un proceso que requiere de entrada de datos en **background** es necesario alimentar su entrada de datos mediante una redireccion por ejemplo:

```
$ ls > foo
```

Creamos un archivo llamado **foo** que contiene los nombres de los archivos existentes en el directorio.

```
$ cat < foo &
```

Redirigimos el archivo **foo** como entrada estandar del proceso **cat** y lo enviamos a **background**.

Una vez enviado el proceso **cat** a **background** con los datos de entrada que requiere, el proceso se ejecuta y nos muestra la salida de su procesamiento en la terminal:

```
$ cat < foo &
[4] 6874
programa1.c
programa2.c
programa3.java
applet.java

[4] Done        cat < foo
```

Tambien nos muestra que el proceso ha finalizado (**[4] Done**) y el nombre de proceso que se encontraba en background (**cat < foo**).

Si no deseamos que la salida del proceso aparezca en la terminal, tambien es posible redirigir su salida a un archivo de la siguiente manera:

```
$ cat < foo > salida &
[4] 6878
[4] Done        cat < foo > salida
```

En donde este comando **cat** lee su entrada de datos requerida (**stdin**) del archivo **foo** y envia su salida de datos (**stdout**) a el archivo llamado salida, adicionalmente este proceso se ejecutara en **background**.

Si el proceso llegara a enviar algun error, este error todavia aparecera en la pantalla debido a que no se ha redirigido la salida de error estandar (el descriptor de archivo numero **2**) para redirigirla podemos hacer:

```
$ cat < foo > salida 2> errores &
```

Un caso tipico para redireccionar la salida de error estandar es el comando **find**.

El comando **find** permite buscar archivos desde una ruta especifica y comparando diferentes caracteristicas de los archivos.

Por ejemplo:

Si desearamos buscar desde el directorio raiz todos los archivos que su nombre fuera: programa1.c entonces ejecutariamos la siguiente linea:

```
$ find / -name programa1.c
```

El comando **find** comenzara desde el directorio raiz (**/**) del sistema a buscar el archivo llamado **programa1.c** entrara a cada subdirectorio del sistema y en las partes que encuentre el archivo, desplegara la ruta en la salida estandar.

Uno de los problemas es que si ejecutamos este comando como un usuario normal (no como el super usuario **root**) es posible que no tengamos permisos para entrar a multiples directorios del sistema, y debido a eso el comando **find** deplegara el siguiente mensaje de error cada vez que trate de entrar a un directorio al cual no tiene permiso:

```
find: /sbin: Permission denied
```

Lo cual significa que el usuario con el cual se ejecuto este comando no tiene permiso para entrar a el directorio (**/sbin** en este caso).

Pero aun asi el comando continua con su funcion de buscar en los directorios a los que si puede entrar.

Dependiendo del tamaño del disco, del numero de archivos y directorios del disco y de la velocidad de la maquina, el comando **find** podria tardar un buen rato en recorrer todo el disco en busca de todos los archivos que coincidan con la caracteristica que solicitamos (realmente no recorre todo el disco, busca en la tabla de directorio e inodos, pero esa es otra historia).

Mientras el comando **find** se este ejecutando en **foreground** no tendremos posibilidades de ejecutar otro comando; tendremos que esperar hasta que finalice su funcion.

Si enviamos el comando **find** a **background** redirigiendo sus salidas y error estandar podriamos hacer algo asi:

```
$ find / -name archivo1.c > encontrado 2> errores &
```

Indicando que la salida estandar del comando **find** (las rutas en donde encontro el archivo llamado **archivo1.c**) se almacene en el archivo con nombre encontrado y que si existe algun error en la ejecucion del comando (no tiene permisos para entrar a buscar en algun directorio) este error se almacene en el archivo **errores**; adicionalmente enviamos el comando a **background** para poder hacer uso de la terminal mientras que el comando realiza su proceso.

Moviendo un proceso de foreground a background

En adicion a ejecutar un proceso en background utilizando el caracter **&** al final de la linea de comandos, se puede mover un proceso que se ejecute en **foreground** hacia **background**. Mientras el proceso en **foreground** esta ejecutandose, el shell no puede ejecutar ningun nuevo comando. Antes de que se puedan introducir nuevos comandos, se requiere supender el proceso en **foreground** (ponerlo en pausa) para obtener nuevamente un prompt de shell. La combinacion de teclas que envian la señal para suspender un proceso en la mayoria de los sistemas Unix es **Control-Z**.

Cuando un proceso en **foreground** es suspendido, el prompt de comando del shell permite introducir nuevos comandos; el proceso original (que fue suspendido) todavia se encuentra en memoria, pero no se le esta permitiendo obtener tiempo de CPU (y por lo cual no procesa ningun dato). Para continuar el proceso en **foreground** (remover la pausa), existen 2 opciones: Continuar en **background** o continuar en **foreground**.

El comando **bg** permite continuar con el proceso suspendido en **background**, mientras que el comando **fg** continua procesando en **foreground**.

Por ejemplo:

Comenzamos con un proceso que tarda algun tiempo en completarse:

(A falta de mejor ejemplo)

```
$ sleep 500
```

Una vez que el proceso esta siendo ejecutado, no podemos ejecutar ningun otro proceso.

Para poder realizar una pausa en el proceso de **sleep**, tecleamos **Control-Z**

```
$ sleep 500
^Z
[6]+  Stopped                 sleep 500

$
```

el proceso **sleep** se detiene y el shell nos muestra un prompt, en este prompt podemos definir si el proceso que acabamos de detener continuara su ejecucion en **foreground** o **background**.

Para enviar el proceso a **background**, utilizamos el comando **bg**:

```
$ sleep 500
^Z
[6]+  Stopped                 sleep 500
$ bg
[6]+ sleep 500 &
```

Ahora el proceso **sleep** esta continuando su ejecucion en **background**.

Por default el comando **bg** mueve a **background** el proceso mas recientemente suspendido. Puede ser el caso de que se tengan multiples procesos suspendidos a la vez. Para diferenciarlos utilizamos el numero de trabajo del shell (el que esta entre corchetes, en el caso anterior es **[6]**) con el caracter % como prefijo.

En el siguiente ejemplo, comenzamos con dos procesos que tardan bastante tiempo en completarse.

```
$ proceso_que_tarda_mucho1
Control-Z
[7]+  Stopped                  proceso_que_tarda_mucho1
$ proceso_que_tarda_mucho2
Control-Z
[8]+  Stopped                  proceso_que_tarda_mucho2
$
```

Para mover el primer proceso a **background** usamos:

```
$ bg %7
[7] proceso_que_tarda_mucho1&
$
```

Para mover el segundo proceso a **background**:

```
$ bg %8
[8] proceso_que_tarda_mucho2&
$
```

La capacidad de poder indicar sobre que proceso realizar una accion especifica (moverlo a **background** o a **foreground**, por ejemplo), muestra la importancia de tener numeros de trabajo asignados.

Moviendo un proceso en background a foreground

Con el comando **fg** es posible mover a foreground el proceso mas recientemente suspendido o movido a background. Tambien se puede especificar el numero de trabajo del shell que se desea mover a **foreground**.

```
$ cat &
[3] 23550
$
[3]+  Stopped                 cat
$
```

El proceso **cat** esta detenido debido a que requiere entrada de datos. Con el comando **fg** es posible enviarlo nuevamente a **foreground** e introducir los datos que necesita:

```
$ fg %3
cat
hola
hola
Control-D

$
```

Si se tienen multipes procesos ejecutandose en background o suspendidos, es posible listarlos con el comando jobs:

```
$ sleep 100 &
[2] 23542
$ jobs
[1]+  Running                 sleep 500 &
[2]+  Running                 sleep 100 &

$
```

Y de esta manera seleccionar el numero de trabajo (encerrado entre corchetes) que se desea manipular .

Esperando que los procesos en background terminen

Existen dos maneras de esperar que un proceso que se ejecuta en **background** termine antes de ejecutar otro comando. Se puede presionar la tecla enter cada cierto tiempo hasta que aparezca el mensaje del shell indicando que el comando se ha completado, o se puede utilizar el comando **wait**.

Existen 3 maneras de utilizar el comando **wait**: sin ninguna opcion, con un numero de proceso (**pid**) o con un caracter % seguido del numero de trabajo del shell.

Si no se especifica el numero de proceso o trabajo, el comando wait espera a que todos los procesos en **background** terminen. Usar **wait** sin ninguna opcion es util en programa de shell que inicia una serie de procesos en **background** y tiene que esperar que termien dichos procesos antes de continuar.

```
$ sleep 10 &
[1] 23705
$ wait
[1]+  Done                    sleep 10
$
```

El una vez que se ha introducido el comando **wait**, el shell no regresa el prompt hasta que los procesos en **background** han concluido.

Tambien es posilbe esperar que un solo proceso ejecutandose en **background** termine, especificando su numero de trabajo asignado por el shell

```
$ wait %1
```

Enviando señales a los procesos

Otro comando para manipular los trabajos del shell y los procesos es el comando **kill**, como su nombre sugiere sirve para "matar" o finalizar un proceso.

De igual manera que los comandos **fg** y **bg**, se puede especificar un numero de trabajo anteponiendo el simbolo %:

```
$ jobs
$ sleep 300 &
[1] 23858
$ kill %1
[1]+  Terminated              sleep 300
```

Tambien se puede utilizar el comando kill para terminar un proceso especificando su numero de proceso asignado por el sistema (**pid**):

```
$ jobs
$ sleep 300 &
[1] 23859
$ kill 23859
[1]+  Terminated              sleep 300
```

En realidad, el comando **kill** no termina o "mata" el proceso; envia a el proceso una señal. Por default envia la señal **15** (**SIGTERM**). Un proceso puede elegir ignorar la señal **15** o usarla para comenzar su proceso de terminacion (cerrar archivos, vaciar variables, etc). Si un proceso ignora el comando **kill**, se puede forzar su terminacion enviando la señal **9** (**SIGKILL**), esta señal no puede ser ignorada.

Para mostrar como algunos procesos pueden ignorar la señal de terminacion (**15**) podemos intentar la terminacion de el proceso de shell (**bash**)

```
$ ps
  PID TTY          TIME CMD
23510 pts/1    00:00:00 bash
23896 pts/1    00:00:00 ps

$ kill 23510
```

bash ignora la señal **15** pero si se envia:

$ kill -9 23510

El proceso de shell sera eliminado (con sus respectivas consecuencias).

Otra señal util para enviar a los procesos es la señal numero **1** (**HUP** o **Hang up**), esta es utilizada por algunos procesos para leer nuevamente sus archivos de configuracion y comenzar nuevamente su proceso.

Filtros de texto

El comando head

Los programas en shell comunmente son creado para reformatear la salida de datos de algunos comandos. Algunas veces esta tarea es facil, tal como desplegar alguna parte de la salida sin incluir algunas lineas, pero en algunos casos este proceso es mas sofisticado.

En este capitulo vermos algunos comandos que son utilizados como filtros de texto en programacion shell, uno de esos comandos es el comando **head**.

La sintaxis basica para el comando **head** es:

```
head -n lineas archivos
```

Donde **archivos** es la lista de archivos que se desea que procese el comando **head**. Si no se especifica ningun archivo, **head** espera entrada de datos de la entrada estandar (**stdin**). El parametro **-n** sirve para especificar el numero de **lineas** a desplegar contando desde el principio del archivo o de la entrada estandar.

Por ejemplo para desplegar las primeras **10** lineas del comando **ps** ejecutamos:

```
$ ps -ef | head
  PID  TT  STAT      TIME COMMAND
    1  ??  Ss     0:00.10 /sbin/init
    2  ??  Ss     0:09.34 /sbin/mach_init
   82  ??  Ss     0:03.99 /usr/sbin/syslogd -s -m 0
   88  ??  Ss     0:10.17 kextd
   90  ??  Ss     5:18.84 /usr/sbin/configd
   91  ??  Ss     0:05.26 /usr/sbin/diskarbitrationd
   96  ??  Ss     0:23.17 /usr/sbin/notifyd
  120  ??  Ss     1:11.34 netinfod -s local
  122  ??  Ss     1:54.93 update
$
```

O para desplegar las primeras **2** lineas del archivo **/etc/hosts**

```
$ head -n 2 /etc/hosts
##
# Host Database
$
```

El parametro **-n** se puede omitir, utilizando solo el numero de lineas precedidas por el caracter **-** (signo de menos)

```
$ head -2 /etc/hosts
##
# Host Database
$
```

Si se especifica mas de un archivo el comando **head** muestra las primeras lineas que se especifiquen de cada archivo mostrando un encabezado que consiste del mensaje

```
==> XXX <==
```

En donde **XXX** representa el nombre de cada uno de los archivos

```
$ head -2 /etc/hosts /etc/passwd
==> /etc/hosts <==
##
# Host Database

==> /etc/passwd <==
##
# User Database
```

El comando tail

Podriamos decir que el comando **tail** realiza la funcion inversa del comando **head**, mientras que **head** despliega **n** numero de lineas de la entrada estandar o de un archivo especificado, el comando **tail** despliega las ultimas **n** lineas de la entrada estandar o del archivo especificado. Su sintaxis basica es:

```
tail -n lineas archivos
```

Donde **lineas** es el numero de ultimas lineas del archivo o entrada estandar a desplegar y archivos especifica el o los archivos a desplegar, si no se especifica un nombre de archivo, **tail** desplegara las ultimas **n** lineas de la entrada estandar, si no se especifica el numero de lineas a desplegar, **tail** desplegara las ultimas 10 lineas del archivo o entrada estandar.

Por ejemplo, para desplegar las ultimas 3 lineas del archivo **/etc/passwd**:

```
$ tail -n 3 /etc/passwd
mysql:*:74:74:MySQL Server:/nohome:/noshell
sshd:*:75:75:sshd Privilege separation:/var/empty:/noshell
unknown:*:99:99:Unknown User:/nohome:/noshell
$
```

De igual manera que con el comando **head**, el parametro **-n** se puede omitir, reemplazandolo por el numero de lineas a desplegar (con un signo de menos antes del numero de lineas)

```
$ tail -3 /etc/passwd
mysql:*:74:74:MySQL Server:/nohome:/noshell
sshd:*:75:75:sshd Privilege separation:/var/empty:/noshell
unknown:*:99:99:Unknown User:/nohome:/noshell
$
```

tail con el parametro **-f** (follow) hace que el comando no se detenga cuando encuentre el fin del archivo a desplegar, en lugar de salir, **tail** espera datos adicionales para desplegarlos. Esta opcion es util para monitorear el crecimiento de algun archivo de forma interactiva.

Combinando los comando **head** y **tail** podriamos extraer una linea especifica de la salida de un comando, por ejemplo:

Digamos que necesitamos saber cual es el ultimo dia del mes, para hacer eso podriamos auxiliarnos del comando **cal**.

El comando **cal** despliega el calendario del mes y año que se le especifique, si no se le especifica un mes o año, despliega el calendario del mes actual:

```
$ cal
     March 2004
 S  M Tu  W Th  F  S
    1  2  3  4  5  6
 7  8  9 10 11 12 13
14 15 16 17 18 19 20
21 22 23 24 25 26 27
28 29 30 31

$
```

Si se le especifica el mes y el año despliega el calendario de dicho mes y año:

```
$ cal 06 04
       June 4
 S  M Tu  W Th  F  S
 1  2  3  4  5  6  7
 8  9 10 11 12 13 14
15 16 17 18 19 20 21
22 23 24 25 26 27 28
29 30
```

Y si se le especifica un año solamente, despliega el calendario completo (de todos los meses de dicho año).

```
$ cal 2002
(desplegara el calendario de todos los meses del año 2002)
```

Regresando a nuestro ejercicio, podriamos ejecutar el comando cal sin ningun parametro para obtener el calendario del mes actual:

```
$ cal
     March 2004
 S  M Tu  W Th  F  S
    1  2  3  4  5  6
 7  8  9 10 11 12 13
14 15 16 17 18 19 20
21 22 23 24 25 26 27
28 29 30 31

$
```

Si deseamos saber cual es el numero del ultimo dia podriamos:

```
$ cal | tail -2 | head -1 | cut -d" " -f4
31
```

Extraemos las utimas 2 lineas de la salida del comando cal, las cuales son:

```
28 29 30 31

$
```

Hay que notar que hay una linea en blanco al final del despliegue, por eso extraemos las ultimas 2, despues de eso tomamos la primera linea, y cortamos el campo numero 4 delimitado por espacios, obteniendo el valor numerico del ultimo dia del mes.

Tengo que mencionar que esta solucion no es eficiente puesto que los meses tienen diferente numero de dias, esto funcionara solo para este mes, una solucion mas funcional se presentara cuando veamos el comando **awk**.

Filtros de texto

El comando grep

El comando **grep**, el cual significa "global regular expresion" (o expresion global regular) permite localizar una expresion en un archivo o en la entrada estandar.

Una expresion puede ser una palabra o una regla de busqueda, comenzaremos buscando palabras para despues buscar reglas especificas de busqueda.

El formato del comando **grep** es:

```
grep expresion archivos
```

Si no se especifica el archivo o archivos en los cuales se buscara la expresion se asume que se buscara en la entrada estandar (**stdin**), para verificar esto tecleamos:

```
$ grep hola
```

Al ejecutar el comando anterior estamos diciendole a **grep** que despliegue unicamente las lineas que contengan la plabra **hola**, debido a que no especificamos ningun archivo, el comando **grep** espera entrada de datos, ahi procedemos a teclear algunos datos y despues la palabra que busca, en este caso **hola**.

```
$ grep hola
asdf
adfasd
hola
hola
Control-D
$
```

Al introducir el texto **hola**, el comando **grep** repite el texto, indicando que la linea anterior comple con la expresion especificada (desplegar las lineas que contengan la palabra **hola**)

Ahora para probar el comando **grep** con un archivo procederemos a crear un archivo con datos simples:

```
$ cat > archivoprueba
hola
adios
foo
bar
datos
nada
Control-D
```

Ya creado el archivo, podemos realizar algunas pruebas:

Mostrar todas las lineas que contengan la palabra **foo**:

```
$ grep foo archivoprueba
foo
```

Mostrar todas las lineas que contengan el caracter "**d**" (en cualquier parte de la linea)

```
$ grep d archivoprueba
adios
datos
nada
```

Ahora buscaremos expresiones, en este caso es util explicar dos de las expresiones mas basicas:

- Un metacaracter es una letra o caracter que tiene un significado especial para el comando, es un caracter especial.
- El metacaracter **^** antes de la expresion indica que la expresion debera de estar al principio de la linea
- El metacaracter **$** despues de la expresion indica que la expresion debera de esta al final de la linea

De esta manera si quisieramos desplegar todas las lineas en las cuales la ultima letra sea la letra "**a**" introduciriamos:

```
$ grep a$ archivoprueba
hola
nada
```

Para desplegar todas las lineas en las cuales la letra inicial (al inicio de la linea) sea el caracter "**a**"

```
$ grep ^a archivoprueba
adios
```

Para que la expresion de busqueda al principio o al final de la linea funcione es necesario que el metacaracter **^** o **$** se encuentren al principio o al final de la expresion respectivamente. Si por alguna razon no se encuentran estos metacaracteres al principio o al final la busqueda no funcionara:

```
$ grep a^ archivoprueba
$
```

En el caso de el caracter **$** (el cual debera ir al final de la expresion) sucede algo interesante si nos equivocamos al poner su posicion, supongamos que deseamos desplegar todas las lineas que al final tengan la letra "**r**" pero nos equivocamos en la posicion del metacaracter **$**:

```
$ grep $r archivoprueba
```

Que sucede y porque ?

Algunos otros metacaracteres son:

El metacaracter **.** (punto) coincide con cualquier caracter, no importa que sea, por lo cual la expresion:

```
r.
```

Coincide con cualquier letra "**r**" seguida de cualquier caracter

La expresion regular

.x.

Coincide con una letra **x** que esta rodeada de dos caracteres no importando cuales sean.

Tambien es posible buscar caracteres especificos con los corchetes, tal como lo hace el shell al definir busquedas de archivos:

La expresion

^[hd]

Coincide con cualquier palabra que comience con la letra **h** o la letra **d**

Por ejemplo:

```
$ grep ^[hd] archivoprueba
hola
datos
```

Tambien se puede especificar un rango de caracteres o letras, separandolas con un signo de menos (**-**):

La expresion:

[A-Z]

Coincide con cualquier letra mayuscula.

La expresion

[A-Za-z]

Coincide con cualquier letra mayuscula o minuscula

Por ejemplo, la expresion:

```
$ grep [a-o]$ archivoprueba
hola
foo
nada
```

Coincide con cualquier linea que termine con cualquier letra desde la **a** hasta la **o**.

Pero que pasa si ponemos el metacaracter **^** dentro de los corchetes:

```
$ grep [^a-o]$ archivoprueba
adios
bar
datos
```

El orden de la expresion se invierte, ahora desplegara cualquier linea en la cual, la ultima letra no sea desde la letra **a** a la letra **o**.

Con lo anterior, si deseamos desplegar todas las lineas que no comiencen con la letra "**a**" introducimos:

```
$ grep ^[^a] archivo prueba
hola
foo
bar
datos
nada
```

El metacaracter * (asterisco) coincide con cero o mas ocurrencias de el caracter que lo precede (que esta antes que el), por lo cual:

```
$ grep adi* archivoprueba
adios
nada
```

Despliega **adios** debido a que el comando grep esta especificando: que contenga la letra **a**, despues la letra **d** y opcionalmente la letra **i** (debido a que el metacaracter * hace que la letra que este antes de el sea opcional) y despliega **nada** puesto que contiene el caracter **a** y **d** juntos y aunque no contiene la letra **i** el metacaracter * indica que no es necesario (cero o mas ocurrencias).

Que realiza el siguiente comando ?

```
$ grep ^d.*s$ archivoprueba
```

Tambien es posible invertir la expresion de el comando **grep** adicionando el parametro **-v**, por ejemplo:

```
$ ps -ef | grep -v root
```

Muestra todas las lineas en las que no aparezca la palabra **root**.

El comando tr

El comando tr singifica "translate" o trasladar y sirve para convertir caracteres de la entrada estandar a la salida estandar, el formato general del comando es:

```
tr original reemplazo
```

En donde **original** y **reemplazo** son uno o mas caracteres. Cualquier caracter especificado en **original** que se encuentre en la entrada estandar, sera convertido a el caracter correspondiente en **reemplazo**.

Por ejemplo:

```
$ echo "abc" | tr a c
cbc
```

En este caso se le indica al comando tr que en donde encuentre el caracter a en la entrada estandar, lo reemplace por el caracter c, por lo cual obtenemos como salida "cbc".

Para cambiar mayusculas a minusculas:

```
$ echo "ABC" | tr [A-Z] [a-z]
abc
```

Para cambiar de minusculas a mayusculas:

```
$ echo "abc" | tr [a-z] [A-Z]
ABC
```

Para reemplazar espacios por puntos:

```
$ echo "esto tiene espacios" | tr " " .
esto.tiene.espacios
```

En algunos casos se requiere eliminar algun caracter o caracteres de la entrada estandar, esto se puede lograr con el parametro **-d** de el comando **tr**, el cual eliminara de la su salida estandar los caracteres especificados, por ejemplo:

```
$ echo "Xeste es el mensajeX" | tr -d X
este es el mensaje.
```

Aunque hay que tener cuidado con esta opcion, debido a que si deseamos eliminar una palabra completa como en el siguiente caso:

```
$ echo "basura, este es el mensaje" | tr -d basura
, ete e el menje
```

obtenemos un resultado no muy deseado, debido a que el comando **tr** elimina no la palabra "**basura**", elimina las letras **b**, **a**, **s**, **u**, **r** y **a**.

Que despliega el siguiente comando ?

```
$ echo "El santos anda OK." | tr O m | tr K a | tr . l
```

El comando sed

Nota: esta es una brevisima intruduccion a el comando sed, se explica solo lo mas basico.

sed significa "stream editor" o editor de entrada de datos, **sed** comunmente es utilizado como un filtro. Lee cada linea de su entrada de datos y entonces realiza las acciones solicitadas. La sintaxis basica de un comando **sed** es:

```
sed 'programa' archivos
```

En donde **archivos** es uno o mas archivos y **programa** es uno o mas comandos de la forma:

```
/expresion/accion
```

En donde **expresion** es una expresion regular y **accion** es uno de los siguientes comandos:

Accion	Descripcion
p	Despliega la linea
d	Elimina la linea
s/dato1/dato2/	Substituye la primera ocurrencia de **dato1** con **dato2**

Comenzaremos con la expresion mas simple, desplegar lineas que coincidan con una expresion:

Para poder realizar unas simples pruebas creemos un archivo simple, digamos una lista de precios de frutas (para variar):

```
$ cat > frutas
Fruta      Precio
Platano    0.89
Pera       0.79
Kiwi       1.50
Manzana    1.29
Mango      2.30
Uva        0.99
Control-D
```

Digamos que deseamos desplegar una lista de las frutas que cuestan menos de 1 peso. Para

eso necesitariamos utilizar el comando **p** de **sed** (**p=print**)

/expresion/p

En donde la expresion seria:

0\.[0-9][0-9]$

Notar como escapamos el punto (**\.**), porque si no lo hacemos significa que despues del 0 puede existir cualquier caracter, no es lo que queremos, queremos que exista un punto.

Es decir, sed debera desplegar todas las lineas en donde se encuentre el numero **0**, punto (**.**), cualquier combinacion de numeros de **0** a **9** y nuevamente cualquier combinacion de numeros de **0** a **9** al final de la linea. Asi el valor podria ser:

```
0.32
0.99
0.45
```

Completando entonces el comando, seria:

```
$ sed '/0\.[0-9][0-9]$/p' frutas
Fruta      Precio
Platano    0.89
Platano    0.89
Pera       0.79
Pera       0.79
Kiwi       1.50
Manzana    1.29
Mango      2.30
Uva        0.99
Uva        0.99
```

Lo que podemos ver es que las lineas que coinciden con nuestra expresion aparecen repetidas en el listado, este es el comportamiento normal del comando **sed**, despliega cada linea de su entrada estandar en la salida estandar. Para evitar este comportamiento podemos especificar el parametro **-n** de la siguiente manera:

```
$ sed -n '/0\.[0-9][0-9]$/p' frutas
Platano    0.89
Pera       0.79
Uva        0.99
```

Eliminando lineas.

Digamos que ya se acabaron los mangos y necesitamos borrarlos de la lista, para esto usaremos el comando **d** (delete) de **sed**, la sintaxis seria la siguiente:

/expresion/d

Y necesitariamos especificar en la expresion que buscara la palabra mango (pero tambien puede ser Mango con una **M** mayuscula!) al principio de la linea, de la siguiente manera:

/^mango/

Pero para asegurarnos

/^[Mm]ango/

Completando el comando:

```
$ sed '/^[Mm]ango/d' frutas
Fruta      Precio
Platano    0.89
Pera       0.79
Kiwi       1.50
Manzana    1.29
Uva        0.99
```

Esta vez no tuvimos que especificar el parametro **-n**, debido a que la opcion **p** (print) imprime toda su entrada de datos a la salida estandar, pero el comando **d**, simplemente elimina de la salida estandar los datos que coincidan con la expresion.

Hay que darnos cuenta que este comando no modifica al archivo original, asi que para modificarlo podriamos:

```
$ sed '/^[Mm]ango/d' frutas > frutas
```

Pero no!, no lo hagas..

Porque ?

Porque el shell primero "blanqueara" el archivo **frutas** (debido a el comando **>**) antes de ejecutar el comando **sed**; al crear o "blaquear" primero el archivo **frutas**, ya el comando **sed** no tendra datos para procesar (debido a que el archivo **frutas** esta vacio!!).

Para hacer la actualizacion podriamos utilizar el shell:

```
$ cp frutas frutas.$$
$ sed '/^[Mm]ango/d' frutas.$$ > frutas
$ rm frutas.$$
```

Nota: `$$` es una variable del shell que contiene el numero de proceso (**`pid`**) del mismo shell.

Realizando substituciones con sed

Ahora digamos que nos equivocamos, y que al teclear la lista de frutas, en lugar de **`Uva`** era **`Naranja`**, por lo cual tendremos que reemplazar **`Uva`** con **`Naranja`** en la lista.

Para hacer esto, utilizaremos el comando **`s`** (**`substitute`**) de **`sed`**, su sintaxis es:

```
/expresion1/s/expresion2/expresion3/
```

Donde **`expresion1`**, **`expresion2`** y **`expresion3`** son expresiones regulares. Cuando se utiliza el comando **`s`**, la **`expresion3`** es reemplazada por la **`expresion2`** en cada linea que cumpla con la **`expresion1`**.

Frecuentemente la **`expresion1`** es omitida asi que tambien podriamos escribir:

```
s/expresion1/expresion2/
```

De esta manera el comando **`s`** se ejecutara en todas las lineas.

Para reemplazar **`Uva`** por **`Naranja`** entonces usaremos:

```
$ sed 's/Uva/Naranja/' frutas
Fruta      Precio
Platano    0.89
Pera       0.79
Kiwi       1.50
Manzana    1.29
Mango      2.30
Naranja        0.99
```

Otro ejemplo rapido seria:

Eliminar un conunto de caracteres de un mensaje:

```
$ echo "<HR>hola<HR>" | sed 's/<HR>//g'
hola
```

Realizando reemplazos globales

Consideremos el siguiente mensaje:

aqi hay muchos errores, aqi tambien, porque aqi ? , porque aqi ?

En donde podemos ver que la palabra **aqi** esta mal escrita multiples veces. Tratemos de arreglar esto con **sed**:

```
$ echo "aqi hay muchos errores, aqi tambien, porque aqi ? , porque aqi ?" | sed 's/aqi/aqui/'
```

Lo cual producira la siguiente salida:

aqui hay muchos errores, aqi tambien, porque aqi ? , porque aqi ?

Solo reemplazo la primera coincidencia, no todas, debido a que este es el comportamiento normal del comando **sed**, si se desea que todas las ocurrencias de la expresion sean reemplazadas se requiere especificar a el comando **sed** que el reemplazo sea global (**g**).

```
$ echo "aqi hay muchos errores, aqi tambien, porque aqi ? , porque aqi ?" | sed 's/aqi/aqui/g'
```

Reusando el valor de la expresion.

Supongamos que tenemos el siguiente texto:

Precio 10.4

y deseamos agregar el signo de pesos a **10.4**, podriamos utilizar el metacaracter **&** que en **sed** almacena el valor que coincide con la **expresion1**, por ejemplo:

```
$ echo "hola" | sed 's/hola/&/'
```

Desplegara "**hola**" debido a que la **expresion1** es "**hola**" y el metacaracter **&** almacena el mismo valor que la **expresion1**. Es decir, la linea anterior seria igual que escribir:

```
echo "hola" | sed 's/hola/hola/'
```

Utilizando esta caracteristica podemos:

```
$ echo "precio 10.4" | sed 's/[0-9][0-9]\.[0-9]$/\$ &/'
Precio $ 10.4
```

AWK

Filtrando texto con awk

El comando awk es un lenguaje de programacion completo que permite buscar expresiones en multiples archivos y condicionalmente modificarlos, leer lineas o cerrar archivos. esta encontrado en todos los sitemas Unix y es muy flexible. El nombre awk viene de los apellidos de sus creadores: Alfred Aho, Peter Weinberger y Brian Kernighan.

Nos concentraremos en los elementos de **awk** que estan mas comunmente encontrados en los programas en shell, especificamente estas caracteristicas son:

- Edicion de campos
- Variables
- Control de flujo

La sintaxis basica para un comando de awk es:

```
awk 'programa' archivos
```

En donde archivos es uno o mas archivos y programa es uno o mas comandos, de la forma:

```
/expresion/acciones
```

Donde expresion es una expresion regular y acciones es un o mas comandos que veremos mas adelante, si la expresion es omitida, awk realiza la accion introducida en cada archivo de entrada.

Veamos una de los procesos mas sencillos en awk, desplegar las lineas de un archivo, en este caso usaremos el archivo de frutas que teniamos por ahi creado.

```
$ awk '{ print; }' frutas
Fruta      Precio
Platano    0.89
Pera       0.79
Kiwi       1.50
Manzana    1.29
Mango      2.30
Uva        0.99
```

el simbo de punto y coma (;) es requerido por awk para indicarle que el comando ha concluido.

AWK

Una de las caracteristicas agradables de **awk** es que automaticamente divide la entrada de datos en campos. Como ya hemos visto un campo es un conjunto de caracteres que estan separados por uno o mas separadores de campos (tradicionalmente caracteres como espacio, :, &, etc). **awk** toma como separadores de campo por default el tabulador y los espacios, aunque es posible especificar el separador de campos que **awk** utilizara asignando un nuevo valor a la variable de ambiente **IFS**.

Cuando una linea es leida, **awk** separa los campos que identifica en variables numericas consecutivas, asigna el numero **1** para el primer campo, el numero **2** para el segundo campo, el **3** para el tercer campo y asi consecutivamente. Para accesar a los valores de dichas varaibles se utiliza el operador **$**. Por lo cual el primer campo es **$1**. (y el numero de campos totales se almacena en la variable interna **NF**, que veremos mas adelante)

Como un ejemplo simple, tratemos de desplegar el dia y el año que regresa el comando date:

```
$ date | awk '{ print $1 $6 }'
```

No existe separacion entre los campos desplegados, este es el comportamiento por default de el comando **awk**, para desplegar un espacio entre los campos tenemos que poner una coma entre ellos, de la siguiente manera:

```
$ date | awk '{ print $1, $6 }'
```

Adicionalmente se puede dar formato a la salida utilizando el comando printf de awk en lugar de el comando print, de la siguiente manera:

```
$ date | awk '{ printf"%s %s\n", $1, $3; }'
```

El comando printf requiere una secuencia de formato, el metacaracter % define el formato de los datos que seran pasados como parametros (**$1** y **$3** en este caso).

La siguiente tabla muestra los diferentes tipos de formato de datos que el metacaracter % puede soportar:

Letra	Tipo de dato
s	String (texto)
c	caracter
d	Numero decimal
x	Numero hexadecimal
o	Numero octal
e	Numero exponencial de punto flotante
f	Numero de punto flotante
g	Punto flotante compacto

Realizando operaciones especificas en base a expresiones

Digamos que queremos marcar las frutas que cuestan mas de 1 peso con un asterisco en el archivo de frutas que por ahi anda. Esto significa que necesitamos realizar diferentes acciones dependiendo de si la expresion coincide o no.

podriamos hacer un programa en awk con la siguiente sintaxis:

```
'/expresion1/ { accion1 } /expresion2/ {accion2}'
```

```
$  awk  '/[1-9].[0-9][0-9]$/  {print  $1,$2,"*";}  /0.[0-9][0-9]$/
{ print; } ' frutas
Platano    0.89
Pera       0.79
Kiwi 1.50 *
Manzana 1.29 *
Mango 2.30 *
Uva        0.99
```

AWK

Recordamos el shell que obtenia el ultimo dia del mes con el comando tail ?

```
cal | tail -2 | head -1 | cut -d" " -f4
```

Si, ese precisamente.

El problema de este shell es que si el mes tiene mas o menos dias, el numero de espacios entre los dias del mes no sera exactamente 4.

Por ejemplo, con el calendario de otro mes: (en este caso marzo del 2005)

```
cal 03 05| tail -2 |head -1 | cut -d" " -f4
```

No funciona, porque no son exactamente 4 dias los que estan en la ultima linea.

Con **awk** podriamos

```
cal 03 05| tail -2 |head -1 | awk '{ print $NF }'
```

Utilizando la variable interna de awk NF, esta variable contiene el numero de campos que leyo awk en cada linea.

Problemoslo:

```
$ echo "campo1 campo2" | awk '{ print NF }'
2
```

Nuevamente:

```
$ echo "1 2 3 4 5 6       7" | awk '{ print NF }'
7
```

Vemos que no importa el numero de espacios o tabulaciones entre los campos, awk los ignora.

En estos ultimos 2 ejemplos estamos imprimiendo el numero de campos que lee awk de la linea de entrada que se le proporciona, para desplegar el ultimo campo, simplemente adicionamos el metacaracter $ a la variable NF

por ejemplo:

```
$ echo "campo1 campo2" | awk '{ print NF }'
2
$ echo "campo1 campo2" | awk '{ print $NF }'
campo2
```

En el primer ejemplo desplegamos el numero de campos (**2**), en el segundo ejemplo, al adicionar el metacaracter **$** a la variable **NF** da el mismo resultado que si hubieramos tecleado:

```
$ echo "campo1 campo2" | awk '{ print $2 }'
campo2
```

Debido a que la variable **NF** contiene el valor **2**.

Nota: falta mucho mas de **awk**!!.

Control de flujo

En este capitulo explicaremos algunos elmentos basicos para poder realizar toma de desiciones dentro de un shell script.

El comando **test**.

La herramienta **test**, evalua una expresion, y si el resultado es verdadero, regresa el status de cero (**0**), si es falsa regresa el status **1**. Si no existe ninguna expresion, **test** regresa el status **1** (falsa)

El formato de el comando test es el siguiente:

test expresion

donde la expresion tiene esta forma simple:

valor1 operador valor2

valor1 y **valor2** pueden ser una variable, salida de datos de un comando o cadenas.

operador tiene que ser un operador logico, tal como: **= > < !=** y operadores de comparacion adicionales.

Por ejemplo:

El comando **test** regresa el valor de estatus (**$?**) cero si la variable **c** contiene los caracteres hola:

```
$ c="hola"
$ test $c = "hola"
$ echo $?
0
```

Ahora probemos con un valor que sabemos sera falso

```
$ test $c = "lalala"
$ echo $?
1
```

Una cuestion importante es que debe de existir un espacio entre cada uno de los elementos, si no es asi, solo nos estamos haciendo como el tio lolo, porque:

```
$ test $c="novalido"
$ echo $?
0
```

Segun el comando **test**, se le esta pasando un solo parametro, el requiere 3, recordemos que el sistema operativo separa los parametros mediante los espacios. Por lo cual

```
$ test $c = "novalido"
$ echo $?
1
```

Cuando se realizan comparaciones de valores alfanumericos, es buena idea el encerrar las variables y los valores a comparar entre comillas, por la siguiente razon:

```
$ c=[Enter]
$ test $c = "hola"
-bash: test: =: unary operator expected
```

El error surge porque la variable **c** no contiene ningun valor, por lo cual el shell solo ve 2 parametros, al encerrarlo en comillas:

```
$ test "$c" = "hola"
```

Aunque el valor de la variable **c** este vacio, se compara ahora una cadena vacia `""` con la palabra **hola**.

Operadores alfanumericos

Operador	Regresa verdadero si:
valor1 = valor2	valor1 es identica a valor2
valor1 != valor2	valor1 no es identico a valor2
valor	valor1 no esta vacio
-n valor	valor no es nulo
-z valor	valor es nulo

Haciendo algunas pruebas:

```
$ dia="lunes"
$ test "$dia" = "lunes"
$ echo $?
0
$ test "$dia" != "martes"
$ echo $?
0
$ test "$dia" = "martes"
$ echo $?
1
```

El comando **test** tiene un conjunto de operadores para realizar comparaciones numericas, la siguiente tabla muestra estos operadores:

Operador	Regresa verdadero si:
`entero1 -eq entero2`	entero1 es igual a entero2
`entero1 -ge entero2`	entero1 es mayor o igual que entero2
`entero1 -gt entero2`	entero1 es mayor que entero2
`entero1 -le entero2`	entero1 es menor o igual que entero2
`entero1 -lt entero2`	entero1 es menor que entero2
`entero1 -ne entero2`	entero1 no es igual a entero2

Realizando algunas pruebas:

```
$ test 5 -gt 3
$ echo $?
0
$ test 5 -gt 10
$ echo $?
1
$ test 10 -eq 10
$ echo $?
0
$ test "hola" -eq "hola"
-bash: test: hola: integer expression expected
```

Pero si es posible:

```
$ val=10
$ test "$val" -gt 3
$ echo $?
0
$ test "$val" -gt 100
$ echo $?
1
$ test "$val" -gt "3"
$ echo $?
0
```

El formato altenativo

El comando **test** es utilizado muy frecuentemente por los programas de shell, por lo cual a alguien se le ocurrio abreviarlo para hacer las cosas mas faciles y legibles:

```
$ test 5 -gt 3
```

es igual a:

```
$ [ 5 -gt 3 ]
```

Tip: busca un archivo llamado **test** en el directorio **/bin**, y tambien busca un archivo llamado **[** en el mismo directorio, cual es la diferencia entre estos archivos ? (quien dijo que los nombres de los programas deberian de ser alfanumericos ?)

Operaciones con archivos

Operador	Regresa verdadero si:
-d archivo	**archivo** es un directorio
-e archivo	**archivo** existe
-f archivo	**archivo** es un archivo ordinario
-r archivo	**archivo** es leible por el proceso
-s archivo	**archivo** tiene longitud mayor a cero
-w archivo	**archivo** es escribible por el proceso
-x archivo	**archivo** es ejecutable
-L archivo	**archivo** es una liga simbolica

El comando

```
[ -d /etc ]
```

Prueba si el directorio **/etc** existe, regresa el valor **0** en la variable **$?** si **/etc** es realmente un directorio.

```
[ -w /etc/profile ]
```

Prueba si el archivo **/etc/profile** existe y el proceso tiene permisos de escritura

El comando **test**, puede asociarse de manera mas facil con el comando **if**.

formato del comando **if**

```
if expresion
then
   sentencias
else
   sentencias
fi
```

Abrimos un editor de textos (como **vi**) y almacenamos un archivo con el siguiente contenido:

```
#!/bin/bash
# mi primer programa en shell

archivo="archivo_no_existe"

if [ -f $archivo ]; then
    echo "el archivo $archivo no existe"
else
    echo "el archivo $archivo si existe"
fi
```

Almacenamos el archivo, digamos con el nombre **mishell.sh** y tenemos 2 opciones para ejecutarlo:

- ejecutar: **sh mishell.sh**
- ejecutar: **chmod 500 mishell.sh y luego ./mishell.sh**

Operadores logicos

El operador logico de negacion **!** puede ser puesto antes de la expresion para negar su evaluacion, por ejemplo:

```
[ ! -r "/etc/shadow" ]
```

o

```
[ ! -f /etc/noexiste ]
```

Tambien puede aplicarse a

```
[ ! "$x1" = "$x2" ]
```

El operador logico AND es representado por la opcion **-a** y va enmedio de 2 expresiones:

```
[ -f "/etc/hosts" -a -r "/etc/group ]
```

o

```
[ 5 -gt 3 -a 10 -lt 100 ]
```

o

```
[ "$c" = "hola" -a "$a" = "adios ]
```

El operador logico OR

Es representado mediante la opcion **-o** y de igual manera que el operador **-a** va enmedio de dos expresiones

```
[ "$c" = "hola" -o "$c" != "no" ]
```

El operador **-o** tiene menos precedencia logica que el operador **-a**, lo cual significa que la expresion:

```
[ "$a" -eq 0 -o "$b" -eq 2 -a "$c" -eq 10 ]
```

Es evaluada por el comando **test** de la siguiente manera:

```
"$a" -eq 0 -o ("$b" -eq 2 -a "$c" -eq 10)
```

Se pueden utilizar parentesis para cambiar el orden de evaluacion, pero se tienen que "escapar" para que no sean interpretados por el shell:

```
[ \( "$a" -eq 0 -o "$b" -eq 2 \) -a "$c" -eq 10 ]
```

Control de flujo

Ciclos de control

Los dos tipos principales de ciclos son:

- el ciclo **`while`**
- el ciclo **`for`**

El ciclo **`while`** permite ejecutar un conjunto de comandos de forma repetida hasta que una condicion ocurra.

El ciclo **`for`** permite ejecutar un conjunto de comandos de forma repetida por cada elemento en una lista.

El cilclo while

La sintaxis basica del ciclo **`while`** es:

```
while comando
do
     sentencias
done
```

Donde comando es normalmente una expresion del comando **`test`** (aunque tambien puede ser un comando para ejecutar).

`sentencias` se refiere a el cuerpo del ciclo **`while`** y contiene el codigo a ejecutar en cada iteracion del ciclo. Las sentencias **`do`** y **`done`** son utilizadas por el comando **`while`** para saber en que partes inicia y termina el ciclo.

La ejecucion de un ciclo while es de la siguiente manera:

1.- se ejecuta el **`comando`**
2.- si el codigo de salida de el **`comando`** es diferente de cero, el ciclo no se ejecuta
3.- si el codigo de salida del **`comando`** es cero, las **`sentencias`** de el cuerpo del ciclo se ejecutan
4.- se regresa al paso numero 1

Si tanto el comando como las sentencias son pocas o muy cortas se puede codificar de la siguiente manera:

```
while command; do sentencias; done
```

Un pequeño ejemplo que usa el ciclo **while** para desplegar numeros del cero al nueve

```
x=0
while [ $x -lt 10 ]
do
    echo $x
    x=$((x+1))
done
```

Cada vez que el ciclo se ejecuta, se verifica si la comparacion inicial **[$x -lt 10]** es verdadera, en caso de que no sea verdadera, el ciclo termina.

Tambien se pueden crear comandos **while** anidados:

```
while comando1
do
    sentencias1
    while comando2
    do
        sentencias2
    done
    sentencias3
done
```

Como un ejemplo podriamos ejecutar este codigo:

```
x=0
while [ "$x" -lt 10 ]
do
    y="$x"
    while [ "$y" -ge 0 ]
    do
        echo -n "$y "
        y=$((y - 1))
    done
    echo
    x=$((x+1))
done
```

El ciclo for

A diferencia del ciclo **while**, el cual sale del ciclo cuando una condicion es **falsa**, el ciclo **for** opera en base a una lista de datos. El ciclo **for** repite un set de comandos por cada dato en la lista.

La sintaxis basica del ciclo **for** es:

```
for variable in dato1 dato2 dato3 ... datoN
do
     sentencias
done
```

Dondo **variable** es el nombre de una variable y **dato1** a **datoN** son secuencias de caracteres o numeros separados por espacios. Cada vez que el ciclo **for** se ejecuta el valor de cada **dato** en la lista es almacenado en la **variable**.

Lo cual significa que las veces que el ciclo for ejecutara las sentencias depende de el numero de palabras, datos o sentencias especificados en la lista, por ejemplo:

Si se especificara la siguiente lista de datos en un ciclo **for**:

```
aveces no hay nada
```

el ciclo se ejecutaria 4 veces.

Por ejemplo:

```
for x in 1 2 3 4 5 6 7 8
do
   echo $x
done
```

o incluso

```
for archivos in /directorio/*
do
   cp $archivos /tmp/respaldo
   chmod 777 /tmp/respaldo/$archivos
done
```

o

```
for x in `cat /etc/hosts`
do
    echo "encontre este dato >$x<"
done
```

o

```
for x in ${array[@]}
do
   echo "valor: $x"
done
```

Control de ciclos

Cuando revisamos el comando **while** ateriormente, vimos que el ciclo terminaba cuando una condicion particular se cumplia.

Pero si por alguna razon la condicion nunca se cumple, el ciclo continuara para siempre. Por ejemplo en el siguiente codigo:

```
x=0
while [ x -lt 10 ]
do
    echo $x
    x=$((x+1))
done
```

En donde en la comparacion nos falto poner metacaracter (**$**) en la variable **x**

Pero aveces es util crear ciclos infinitos controlados por las sentencias del ciclo, por ejemplo:

```
while :
do
   x=$((x+1)
   if [ $x -gt 100 ]; then
        break;
   fi
done
```

El operador **:** (dos puntos) es un comando del sistema que su valor de ejecucion siempre es 0, por lo cual es util para crear ciclos **while** infinitos.

O por ejemplo, para crear un daemon:

```
#!/bin/sh

usuario="administrador@servidor.correo.com"

while :
do
  if [ `ps -ef | grep "sshd" | grep -v grep | wc -l` -lt 1 ]; then
     # el proceso no esta corriendo!!
     echo "aguas!!, el proceso sshd ya no corre" > /tmp/procdaemon.$$
     mail -s"aguas!!"  $usuario < /tmp/procdaemon.$$
     rm /tmp/procdaemon.$$
     # tratando de levantar nuevamente el proceso
     /etc/rc.d/init.d/sshd stop
     /etc/rc.d/init.d/sshd start
   fi
   sleep 300                # 5 minutos
done
```

Entrada y salida de datos en programas de shell

Entrada y salida de datos en shell scripts

Hasta este momento hemos realizado programas sencillos de shell, pero en ninguno permidos datos por parte del usuario.

Para solicitar datos a un usuario (introducir datos) existe el comando **read**.

La sintaxis general del comando **read** es:

```
read lista_de_variables
```

En donde **lista_de_variables** es una o mas variables en donde se almacenada la informacion, estas variables deberan de estar separadas por espacios.

El comando **read** espera entrada de datos de la entrada estandar (**stdin**) y almacena los valores introducidos en las variables o variable especificada.

Por ejemplo:

```
$ read a
10 [enter]
```

Cuando ejecutamos el comando, **read** espera en la entrada estandar que se introduzcan datos, estos datos se almacenaran en la variable llamada **a**.

Tambien podemos pedir multiples valores para multiples variables:

```
$ read a b c
5 3 2 [Enter]
$ echo $a
5
$ echo $b
3
$ echo $c
2

$
```

Pero si solo introducimos un valor:

```
$ read a b c
10 [Enter]
$ echo $a
10
$ echo $b

$ echo $c

$
```

No se almacenan valores en las variables **b** y **c** debido a que no hay datos para introducir (solo se introdujo un valor y no 3).

Normalmente, solo se pide una sola variable, pero para hacer eso aveces es necesario indicarle al usuario que datos introducira.

Hacemos un shell script:

```
#!/bin/bash
echo -e "Introduzca un valor numerico : \c"
read valor
echo "El valor que se introdujo fue: $valor"
```

Y al ejecutarlo:

```
Introduzca un valor numerico : 10 [Enter]
El valor que se introdujo fue: 10
```

Estamos usando el parametro **-e**, el cual indica a el comando **echo** que interprete los caracteres de control que estan dentro del mensaje que especificamos (el caracter de control **\c**). A continuacion desplegamos algunos de los caracteres de control mas utiles en la programacion shell:

Codigo de escape	**Resultado**
\e	Envia un caracter de escape (esc)
\a	Envia un caracter de timbre (bell)
\b	Envia un caracter de retroceso (backspace)
\f	Envia un caracter de avance de forma (form feed)
\n	Envia un caracter de retorno de carro (line feed y CR)
\r	Envia un caracter de avance de linea (LF)
\t	Envia un caracter de tabulacion (Tab)
\v	Envia un caracter de tabulacion vertical
\'	Envia un caracter de coma simple
****	Envia el caracer de diagonal inversa (\)
\c	Mantiene el cursor en la misma linea

Para que el comando **echo** pueda interpretar los comandos anteriores, se requiere especificar la opcion **-e.**

Si solo se desea que no se realice el retorno de carro automatico cada vez que echo despliega un mensaje se puede utilizar la opcion **-n**

Expect

Introduccion a Expect

expect es un programa que "dialoga" con otros programas que requieren intervencion interactiva de acuerdo a un script. Siguiendo el script, **expect** sabe que datos debe de esperar de un programa y que respuesta debera de enviar.

expect incluye in lenguaje interprete que provee condiciones y estructuras de control de alto nivel para determinar el dialog. En adicion, el usuario puede tomar control y interactuar directamente cuando lo desee, pudiendo despues regresar el control al script.

El nombre **expect** viene de la idea de secuencias send/expect (envia/espera) popularizadas hace ya mucho tiempo por programas como **uucp**, **kermit** y otros programas de control de modem, pero a diferencia de **uucp**, **expect** es un programa general que puede ser ejecutado como un comando de nivel de usuario para interactuar con cualquier aplicacion.

Por ejemplo, algunas de las cosas que **expect** puede hacer:

- Entrar automaticamente a otro sistema, realizar comandos en el sistema remoto y salir del sistema remoto
- "Dialogar" con otro tipo de dispositivos que requieren interaccion humana, tales como: ruteadores, switches, pbx, etc.
- Ejecutar programas de unix/linux que no permiten redireccion de entrada estandar

En general, **expect** es util para ejecutar cualquier programa que requiere interaccion del usuario. Todo lo que es necesario, es que la interaccion pueda ser caracterizada programaticamente (es decir, que se pueda realizar un guion de interaccion con el programa). **expect** tambien puede regresar el control de el programa al usuario (sin detener el programa que se esta controlando), y de igual manera el usuario puede regresar el control a el script en cualquier momento.

El formato general de ejecucion de **expect** es el siguiente:

expect -c comandos archivo_de_comandos argumentos

comandos es un comando especifico que **expect** puede ejecutar antes de comenzar a leer el **archivo_de_comandos**, el cual contiene el script "interactivo" a ejecutar. **argumentos** es un conjunto de argumentos opcionales (una lista de argumentos) que se almacenan en la variable interna **argv** de **expect**, cuando esto se introducen argumentos, la variable **argc** de expect incluye la longitud de el array de argumentos.

expect lee un archivo de comandos en el cual se define el script deseado.

Comandos basicos de expect

Los comandos mas normalmente utilizaods de expect son los siguientes:

- **spawn**
- **expect**
- **send**
- **interact**

El comando **spawn**

El formato de el comando **spawn** es el siguiente:

spawn opciones proceso argumentos

En donde **opciones** es algun argumento para el comando **spawn**, **proceso** es un proceso que el comando **spawn** ejecutara y **argumentos** es la lista de argumentos opcionales para el proceso ejecutado por **spawn**.

Cuando el comando **spawn** ejecuta un proceso, la entrada estandar y salida estandar de el proceso es dirigida a **expect**, de esta manera, **expect** podra controlar el proceso.

Por ejemplo:

swpawn telnet mi.server.com

Indica a **expect** que ejecute el comando **telnet** con el parametro **mi.server.com**, este comando (**telnet**) tendra su entrada estandar y salida estandar redirigida a el comando **expect**.

Por default cuando el comando **spawn** ejecuta un proceso, despliega el nombre del proceso y los argumentos pasados al mismo, si no se desea que se realice lo anterior se puede especificar la opcion **-noecho** de la siguiente manera:

spawn -noecho telnet mi.server.com

Si el programa indicado para su ejecucion por **spawn** no puede encontrarse o se encuentra algun problema, un mensaje de error sera regresado por el siguiente comando en el script. Esta es una caracteristica interna de la forma en que el proceso es ejecutado por **expect**.

El comando expect

`expect` espera una cadena o una expresion regular en la entrada estandar del proceso iniciado por el comando **`spawn`**.

Las instrucciones de **`expect`** pueden construirse de 2 maneras basicas, la forma simple y la forma agrupada

Forma simple:

En la forma simple del comando **`expect`**, se indica que caracteres seran esperados antes de continuar con la ejecucion de el script, por ejemplo:

```
spawn telnet mi.servidor.com
expect "Login:"
```

Esperara que el proceso de telnet despliegue la palabra **`Login:`** antes de continuar con el script.

Tambien es posible indicar una expresion regular mediante el comando **`-re`**.

Esto es util especialmente con algunos mensajes de login, por ejemplo un mensaje de login de el comando **`ftp`**:

```
$ ftp 127.0.0.1
Connected to localhost.
220 localhost FTP server (lukemftpd 1.1) ready.
Name (127.0.0.1:demon):
```

En el cual, el prompt de login varia dependiendo de la direccion ip desde donde se este conectando el usuario y tambien del nombre de usuario local, asi que seria una buena idea el ignorar el contenido de los parentesis e indicarle al comando **`expect`** que no los tome en cuenta mediante una expresion regular como la siguiente: (las expresiones regulares no llevan comillas)

```
expect -re  Name (.*):
```

Forma agrupada

En la forma agrupada del comando **expect** se pueden realizar multiples comparaciones de la salida estandar del comando ejecutado mediante **spawn** y tambien se pueden realizar diferentes acciones en base a cada uno de los datos esperados:

```
expect {
     "failed"     { puts "sistema ocupado" }
     "connected"  { puts "sistema conectado" }
}
```

En este ejemplo el comando **puts** despliega el mensaje que le sigue en la salida estandar del usuario que ejecuta el comando.

Por ejemplo:

Almacenamos en el archivo **uno.exp**:

```
spawn bash
expect "$ "
puts "ya estoy adentro"
```

y ejecutamos

```
$ expect uno.exp
spawn bash
ya estoy adentro
```

En el ejemplo anterior, el comando **expect** ejecuta un shell **bash**, y espera que aparezca una linea con el caracter **$** al final de la misma (el prompt), al ya no existir mas comandos en el archivo el script finaliza, eliminando los procesos levantados por el mismo (el **bash** adicional que ha levantado).

Podemos ver que se el comando **expect** despliega el comando **spawn** y el programa que esta iniciando, para evitar lo anterior, modificamos el archivo que hemos creado y adicionamos la opcion **-noecho** a el comando **spawn**:

```
spawn -noecho bash
expect "$ "
puts "ya estoy adentro"
```

Y volvemos a ejecutar el comando para verificar que ya no aparece el comando **spawn.**

El comando send

El comando **send**, como su nombre lo dice permite enviar datos a el proceso levantado por el comando **spawn**, su sintaxis general es:

```
send opciones datos
```

en donde **opciones** es un argumento que puede modificar la forma de enviar datos de el comando **send**, y **datos** son los caracteres o conjunto de caracteres a enviar a el proceso.

Aunque no es necesario, se recomienda que los datos a enviar por el comando **send** sean encerrados entre comillas para evitar conflictos con espacios y caracteres especiales.

Por ejemplo, para enviar la palabra hola usariamos:

```
send "hola"
```

Tradicionalmente el comando **send** es utilizado despues de un comando **expect** (se envian los datos despues de esperar alguna caracteristica especifica, como un prompt o un mensaje de listo para comandos).

Por ejemplo:

```
spawn bash
expect "$ "
puts "ya estoy adentro"
send "cal\r"
expect "$ "
```

Hay que notar el caracter **\r** el cual envia la tecla **[Enter]**.

Al probar el script anterior:

```
ya estoy adentro
cal
     March 2004
 S  M Tu  W Th  F  S
    1  2  3  4  5  6
 7  8  9 10 11 12 13
14 15 16 17 18 19 20
21 22 23 24 25 26 27
28 29 30 31
```

Otros comandos de expect

Otros comandos utiles de expect son**:**

interact

Permite a el usuario interactuar con la aplicacion controlada por **expect**, como en el siguiente script:

```
spawn telnet miserver.server.com
expect "ame:"
send "username03\r"
expect "word:"
send "secret\r"
interact
```

El comando **sleep**

Espera cierto numero de segundos antes de continuar con el script, util cuando el servidor al que conectamos es muy lento.

Por ejemplo:

```
spawn telnet miserver.server.com
expect "ogin:"
send "usernew\r"
sleep 2
expect "word:"
send "supersecret\r"
sleep 2
expect "\% "
send "echo -n \"datos:\";uptime\r"
sleep 2
expect "\% "
send "exit\r"
```

Un ejemplo en expect

Transferir un archivo mediante ftp:

```
spawn -noecho ftp server.address.com
expect -re "(.*):"
send "luser\r"
expect "word:"
send "secret\r"
expect "ftp>"
send  "get myfile.txt\r"
expect "ftp>"
send "quit\r"
```

Tambien se puede redirigir la salida del comando expect a un archivo para despues procesarlo mediante herramientas del shell, tomando eso en cuenta podriamos:

- Almacenar los siguientes comandos de expect en un archivo, digamos entra.exp:

```
spawn telnet miserver.server.com
expect "ogin:"
send "usernew\r"
sleep 2
expect "word:"
send "supersecret\r"
sleep 2
expect "\% "
send "echo -n \"datos:\";uptime\r"
sleep 2
expect "\% "
send "exit\r"
```

- Crear un shell que llame a expect:

```
#!/bin/bash

datos=`expect -f entra.exp | grep datos: | awk '{ print $4 }'`
echo "el servidor lleva $datos dias levantado"
```

Necesitamos tu ayuda.

Tus comentarios y contribuciones acerca de este documento es lo unico que pedimos a cambio.

Por favor dejanos saber tu opinion acerca del mismo.

demon@demon.com.mx

Adrian de los Santos.