Sistemas Operativos

ICC329-1

Profesor: Carlos E. Méndez Dumestre
carlos.mendez@ufrontera.cl

¿Qué es la Shell?

Una ***Shell*** de ***Unix*** o también shell, es el término usado en informática para referirse a un ***intérprete de comandos***, el cual consiste en la interfaz de usuario tradicional de los sistemas operativos basados en Unix y similares como GNU/Linux.
Mediante las instrucciones que aporta el intérprete, el usuario puede comunicarse con el ***kernel*** y por extensión, ejecutar dichas órdenes, así como herramientas que le permiten controlar el funcionamiento de la computadora.
Los usuarios de GNU/Linux, Unix y similares, pueden elegir entre distintos shells (programa que se debería ejecutar cuando inician la sesión, véase bash, ash, csh, Zsh, ksh, tcsh). Las interfaces de usuario gráficas para Unix, como son GNOME, KDE y Xfce pueden ser llamadas shells visuales o shells gráficas. Por sí mismo, el término shell es asociado usualmente con la línea de comandos.

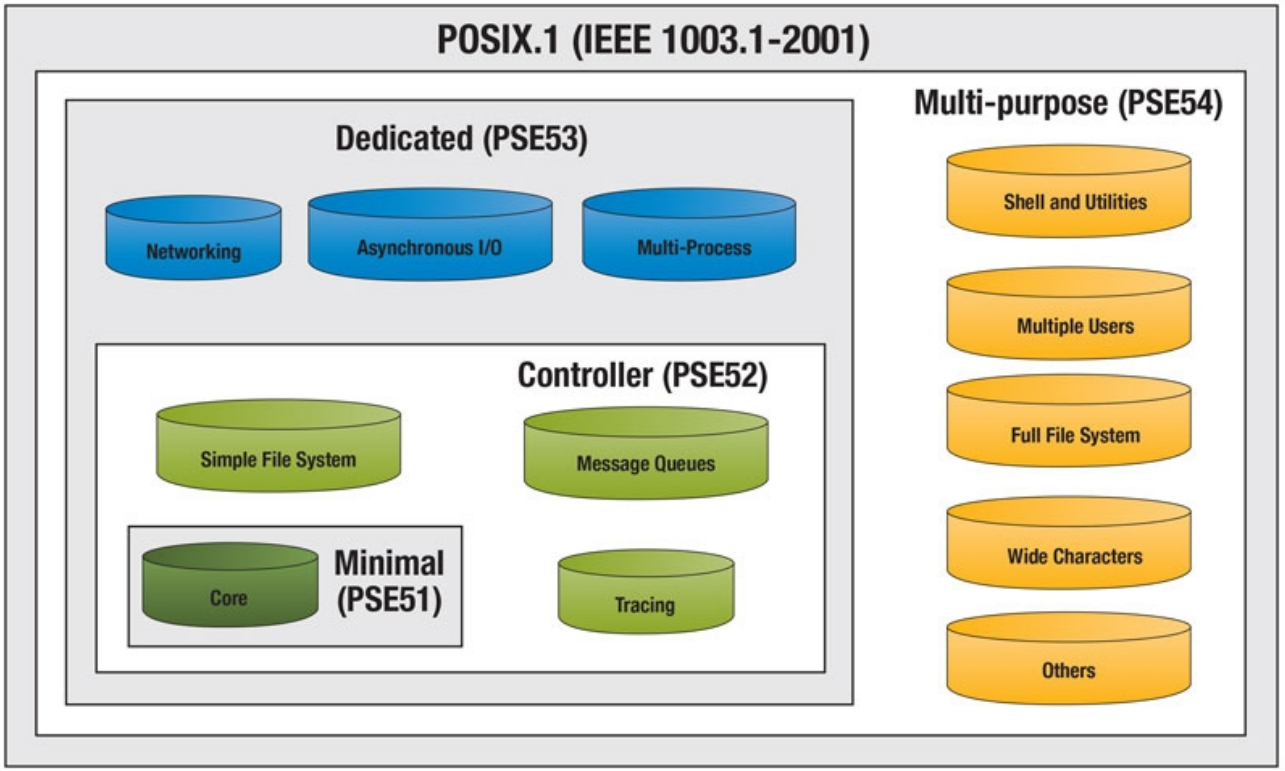
Bourne Shell:
El ***Bourne shell*** fue el shell usado en las primeras versiones de Unix y se convirtió en un estándar de facto; todos los sistemas similares a Unix tienen al menos un shell compatible con el Bourne shell.

Compatibles con Bourne shell:

1. ***Bourne shell (sh)*** -- Escrita por Steve Bourne, cuando estaba en Bell Labs. Se distribuyó por primera vez con la Version 7 Unix, en 1978, y se mejoró con los años.
2. ***Almquist shell (ash***) -- Se escribió como reemplazo de la shell Bourne con licencia BSD; la sh de FreeBSD, NetBSD (y sus derivados) están basados en ash y se han mejorado conforme a POSIX para la ocasión.
3. ***Bourne-Again shell (bash)*** -- Se escribió como parte del proyecto GNU para proveerlo de un superconjunto de funcionalidad con la shell Bourne.
4. ***Debian Almquist shell (dash)*** -- Dash es un reemplazo moderno de ash en Debian.
5. ***Z shell (zsh)*** -- Considerada como la más completa: es lo más cercano que existe en abarcar un superconjunto de sh, ash, bash, csh, ksh, y tcsh.

POSIX

POSIX es el acrónimo de ***Portable Operating System Interface***, y X viene de ***UNIX*** como seña de identidad de la API.
El término fue sugerido por ***Richard Stallman*** en la década de 1980, respuesta a la demanda de la ***IEEE (Institute of Electrical and Electronics Engineers:*** *es una asociación mundial de ingenieros dedicada a la estandarización y el desarrollo en áreas técnicas.****)***, que buscaba un nombre fácil de recordar. La traducción del acrónimo es "Interfaz de Sistema Operativo Portable".
POSIX es una norma escrita por la ***IEEE***. Dicha norma define una interfaz estándar del sistema operativo y el entorno, incluyendo un intérprete de comandos (o "shell"), y programas de utilidades comunes para apoyar la portabilidad de las aplicaciones a nivel de código fuente. El nombre POSIX surgió de la recomendación de Richard Stallman, que por aquel entonces en la década de 1980 formaba parte del comité de IEEE.



Programación en Shell

GNU/Linux, así como la mayoría de los sistemas operativos basados en UNIX, utilizan ***shell scripts*** para realizar una infinidad de tareas. Un shell script es un programa que se escribe con una sintaxis particular, en un archivo de texto plano, para que sea interpretado por un shell, en este caso /bin/bash.

Un shell script es básicamente un ***programa que llama a otros programas***, con la posibilidad de hacer algún tipo de procesamiento propio (como control de flujo, operaciones matemáticas simples, etc.).

La forma de crear un shell script es hacer un archivo de texto plano con cualquier editor de texto disponible en el sistema (vi, nano, emacs, etc.), por ejemplo:

vi hola_mundo

Una vez dentro de la edición del archivo, escribimos las instrucciones que deseamos que ejecute:

```
#!/bin/bash
echo "Hola mundo!"
```

Luego guardamos el archivo y le damos permisos de ejecución:

chmod +x hola_mundo

Y ya podemos ejecutarlo siguiente forma:

./hola_mundo

Este script mostrará por pantalla el mensaje “Hola Mundo!”

> *La primera linea de nuestro script bash: **`#!/bin/bash`**, le indica al sistema que el script será interpretado por el programa que se encuentra a continuación de **#!**, en este caso, **/bin/bash**.*

> *Los comentarios comienzan con # y se extienden hasta el final de la línea. Es muy útil ir comentando el código que uno escribe, para recordar qué realizan ciertas funciones o algoritmos, y otra persona pueda comprender el funcionamiento de nuestro **script**.*

Variables

Las ***variables*** en un script ***BASH*** son simplemente identificadores, sin tipo. Para asignar un valor a una variable, se utiliza el operador ***“=”***, por ejemplo:

```
MIVARIABLE=4
```

Por convención, los nombres de las variables se usan en mayúsculas, aunque no es obligatorio. Para usar el contenido de la variable, dentro de un script, se usa el operador ***“$”***. Por ejemplo:

```
echo $MIVARIABLE
```

Para utilizar el contenido de una variable, seguida de un texto, debemos usar las llaves ”{ }”. Consideremos este ejemplo:

```
#!/bin/bash
#Este script cambia el nombre del archivo tmp a tmp-bak
ARCHIVO="/tmp/ej"
mv $ARCHIVO $ARCHIVO-bak
```

En este caso, el script no funcionaría, pues bash interpretaría a “$ARCHIVO” y “$ARCHIVO-bak” como dos variables distintas, para evitar esto debemos reescribirlo de esta manera:

```
#!/bin/bash
#Este script cambia el nombre del archivo tmp a tmp-bak
ARCHIVO="/tmp/ej"
mv $ARCHIVO ${ARCHIVO}-bak
```

Ahora sí funcionará nuestro script, cuyo único propósito es renombrar el archivo “ej” por “ej-bak”.

Uso de las comillas

En el shell, el “***espacio***”, o el “***tab***”, son separadores. Es decir, que cuando al shell le indicamos

ls -l hola que tal

Lo interpreta como que le pedimos que nos de información sobre tres archivos, llamados: “hola”, “que”, y “tal”. Si en realidad, lo que queríamos, era información sobre un archivo llamado "hola que tal", entonces hay varias maneras de indicarle al shell que los espacios entre esas palabras no deben ser separadores:

Escape (\ )

Hay un caracter de escape, que indica al shell que el siguiente carácter no es especial. Y es la barra invertida. Por lo tanto, podríamos obtener la información del archivo "hola que tal" de la siguiente forma:

ls -l hola\ que\ tal

Los espacios no son especiales (no son separadores), y "hola que tal" es una sola palabra. Por lo tanto, si queremos incluir la **** en alguna parte, entonces debemos ponerla 2 veces (\\), la primera para decirle a BASH que no tome como carácter especial lo que sigue y la segunda como ese carácter que queremos incluír. Esto es muy común en casos como:

cd algún\ directorio\ con\ espacios

Algunos caracteres especiales más:
\ @ ! | < > [ ] { } ( ) ? * $ ' ^ ` " # & ;

Uso de las comillas

Comillas dobles (" ")

Las comillas dobles hacen que los espacios entre las comillas no sean especiales. Por lo tanto, podríamos haber utilizado:

ls -l "hola que tal"

Comillas simples (' ')

Las comillas simples logran que ningún caracter (salvo la comilla simple misma) sea especial. Por ejemplo, si quisieramos crear un archivo que se llame “*@$&”, lo debemos hacer rodeándolo de comillas simples:

touch '*@$&'

Si queremos poner una comilla simple, debemos "escaparla". Para crear un archivo llamado “que'tal”, deberíamos hacerlo así:

touch 'que\'tal'

Ya que si no lo hacemos, la segunda comilla "cierra" la primera.

Comilla invertida (` `)

Las comillas invertidas son más raras. Deben rodear un comando. Ese comando será ejecutado, y lo que ese comando imprima reemplazará al contenido de las comillas invertidas.
Por ejemplo, el comando ls:

```
#!/bin/bash
#Este script ejecuta el comando ls
COMANDO=`ls`
echo $COMANDO
```

De esta manera le indicamos al shell que muestre el resultado del comando “ls” y no que muestre la palabra “ls”.
El uso más frecuente de las comillas invertidas es poder ***asignar el resultado de un comando a una variable***.

Operaciones Aritméticas

Existen varias formas de calcular valores dentro de un ***shell script.*** Tradicionalmente, estos cálculos se hicieron con programas externos, esto generaba un retardo inmenso en la ejecución del shell script.

Hoy los nuevos intérpretes traen la posibilidad de hacer cálculos internamente. Para esto se utiliza una sintaxis especial, y es muy importante que los valores de las variables que utilicen para hacer estos cálculos sean números únicamente.

La sintaxis para hacer operaciones aritméticas es la siguiente:

$[ ]

Las operaciones que se pueden realizar son:

**** suma $[1+1]***
**** resta $[2-1]***
**** multiplicación $[2*2]***
**** división $[2/2]***
**** otras como suma de bits, sacar el módulo, evaluación de igualdad, etc.***

Ejemplo:

```
#!/bin/bash
#
# Operaciones aritméticas
#
x=2
SUMA=$[$x+1]
echo $SUMA
```

Interacción con el usuario

Muchos programas no serían factibles si no tuviéramos algún mecanismo para ***interactuar*** con el usuario, ya sea un simple ***"Presione Enter para continuar"*** o algo más sofisticado, como una lista de opciones de las cuales escoger. Cuando el programa está escrito en shell, es muy sencillo lograr ambas cosas, utilizando dos herramientas: ***read*** y ***dialog***.

read

El comando read es muy sencillo. Le indicamos que pida el valor de una variable al usuario, el usuario escribe una línea de texto (es decir, cualquier cosa hasta que presione “Enter”), y la variable toma el valor que el usuario ingresó.

Por ejemplo:

```
#!/bin/bash
#
#
read TEXTO
echo “El texto ingresado es:”
echo $TEXTO
```

(Si deseamos sólo un "Presione enter para continuar" es exactamente lo mismo, simplemente ignoramos el valor de la variable).

dialog

Dialog es un programa que crea una "***interfaz***" para que el usuario interactúe, y entrega por la salida estándar el resultado de la acción del usuario.

Puede producir: ***preguntas si/no***", "***menú***", "***lista***", "***calendario***", "***barra de progreso***", "***diálogo de contraseña***", "***cuadro de texto***", "***cuadro de mensaje***", etc.

Mediante el comando "*dialog –-help*" se puede obtener más información respecto de éste comando.

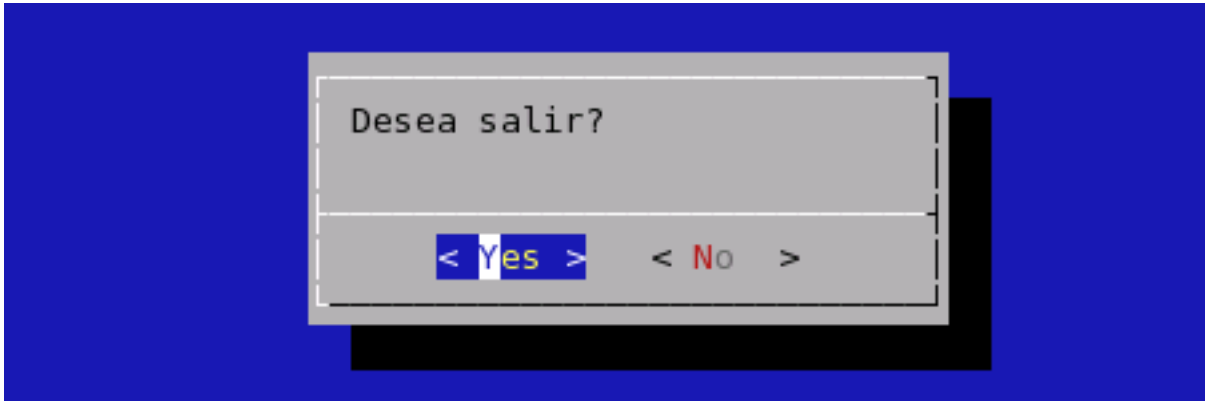
Sintaxis:

dialog { --opciones específicas }

Ejemplo 1: (Diálogo "yesno")

```
#!/bin/bash
#
# Ventana de Dialogo
#
dialog --yesno "Desea salir?" 10 20
```

El argumento "Desea salir?" corresponde al mensaje que se mostrará en la ventana. Los argumentos 10 y 20 corresponden a la altura y ancho, respectivamente, del cuadro.



*Aparte del programa "**dialog**", que produce una salida por consola, puede ser que tenga en su sistema un programa "**gdialog**", "**Xdialog**" o "**dldialog**" (dependiendo de la distribución de GNU/Linux que utilice), que son lo mismo, solo que abren una ventana gráfica, por X11, para el diálogo.*

Ejemplo 2: (Diálogo "inputbox")

El “***inputbox***” funciona básicamente igual que el “***yesno***” pero permite ingresar un texto y capturarlo en una variable.

```
#!/bin/bash
#
# Ventana de Dialogo para ingresar texto
#
NOMBRE=`dialog --inputbox -–stdout "Ingrese su Nombre:" 10 30`
echo “Su nombre es:”
echo $NOMBRE
```

Notar que agregamos el argumento **"*-–stdout*"**, para indicar que el resultado del comando debe ser mostrado por pantalla. Además, el comando debe ir encerrado entre ***comillas invertidas*** para permitir que su valor sea asociado a una variable. (tal como lo mencionamos en un punto anterior)

Ejemplo 3: (Diálogo “Menu”)
Este diálogo despliega un menú con varias opciones, de las que se puede seleccionar.

```
#!/bin/bash
#
# Ventana de Dialogo Menu
#
OPCION=`dialog --menu --–stdout "Seleccione:" 15 40 7
                              1 Opcion1 2 Opcion2 3 Opcion3 4 Opcion4 `
echo “La Op. seleccionada es:”
echo $OPCION
```

Ejemplo 4: (Diálogo “Msgbox”)

Este diálogo despliega un mensaje. Modificando el ejemplo 2, podríamos incluir un msgbox para que nos muestre el texto ingresado:

```
#!/bin/bash
#
# Ventana de Dialogo para ingresar texto
#
NOMBRE=`dialog --inputbox -–stdout "Ingrese su Nombre:" 10 30`
`dialog --msgbox -–stdout "Su Nombre es: $NOMBRE" 10 30`
```