Shell

Enrique Calderón, Francisco Galindo
Estudiante de Ingeniería en Computación

Universidad Nacional Autónoma de México
Facultad de Ingeniería

Curso de SysAdmin, 2025-1

Información del tema

Tiempo estimado

Aproximadamente 120 minutos de clase.

Objetivos

Que los alumnos comprendan que es una shell, como se interactúa con ella además de el uso de jobs para mandar ejecuciones a background, los archivos de configuración, entornos, caracteres especiales y scripting.

Introducción al tema

Una shell es la capa interactiva entre el usuario y el sistema. Es la interfaz que te permite ejecutar comandos, administrar archivos y tener acceso a los recursos del sistema.

Las mas comunes:

- Bash (Bourne-Again SHell): La mas común
- Zsh (Z shell): Shell con autocompletado plugins, temas y mas
- Fish: Shell fácil de aprender
- TCSH: Similar a bash, usado en multiples sistemas unix.

Funciones de una shell

- Interpretar comandos
- Gestionar archivos
- Ejecutar comandos
- Variables de entorno
- Scripting

Jobs

Listar jobs

```
jobs
```

Suspender un comando

```
make all
#<ctrl+z>
```

Correr el comando en background

```
bg
```

Mandar un proceso a background

```
make all &
```

Traer un proceso de regreso

```
fg
```

Entorno

Listar todas las variables en el entorno actual

`env`

$PATH

Es el lugar donde la shell buscará los archivos ejecutables que le presentará al usuario sin necesidad de la ruta absoluta.

Archivos importantes

~/.bash_profile y ~/.bashrc

Archivos de configuración de arranque. Se ejecutan al iniciar una shell.

~/.bash_history

Almacena el historial de comandos. Se puede pedir imprimir a stdout

```
history
```

O se puede buscar en el historia con `<ctrl+r>`

Caracteres a recordar

Character	Name(s)	Uses
*	star, asterisk	Regular expression, glob character
.	dot	Current directory, file/hostname delimiter
!	bang	Negation, command history
|	pipe	
/	(forward) slash	Directory delimiter, search command
\	backslash	Literals, macros (never directories)
$	dollar	Variables, end of line

Caracteres a recordar

Character	Name(s)	Uses
'	tick, (single) quote	Literal strings
`	backtick, backquote	Command substitution
"	double quote	Semi-literal strings
^	caret	Negation, beginning of line
~	tilde, squiggle	Negation, directory shortcut
#	hash, sharp, pound	Comments, preprocessor, substitutions

Caracteres a recordar

Character	Name(s)	Uses
[ ]	(square) brackets	Ranges
{ }	braces, (curly) brackets	Statement blocks, ranges
_	underscore, under	Cheap substitute for a space used when spaces aren't wanted or allowed, or when autocomplete algorithms get confused

Scripting

De la misma manera en que puedes escribir comandos directamente en una *shell* para que sean ejecutados, estos también pueden ser leídos a partir de archivos.

Así, un *script shell* es un archivo que contiene una serie de comandos que pueden ser ejecutados rápidamente por la *shell* sin la intervención directa de una persona

¿Cuándo crear un *script*?

En general, es una buena idea crear un *script de shell* si se desea automatizar la ejecución de una serie de comandos, y que organizar la ejecución de estos comandos no se necesite una alta computacional. Es decir, cuando se quiera evitar hacer tareas engorrosas, especialmente relacionadas con archivos en el sistema de archivos.

Si es necesario hacer muchos cálculos aritméticos, o es necesario manipular estructuras de datos de forma eficiente, será una mejor idea escribir un programa en algún otro lenguaje mejor preparado para ello (C, Python, . . .)

Creando un *script*

Se crea un archivo, con el nombre `primerScript.sh` por ejemplo, con los siguientes contenidos

```
#!/bin/bash

echo "Hola, mundo"
```

La parte de texto que dice `#!` se llama *shebang*, sirve para indicarle al sistema operativo qué programa se va a encargar de interpretar y ejecutar el archivo, en caso de que se pueda ejecutar.

Creando un *script*

Si ahora en tu shell ejecutas:

```
$ bash primerScript.sh
Hola, mundo
```

El comando anterior ejecuta la shell `bash` y le indica que debe interpretar el archivo `primerScript.sh`

Creando un *script*

En su lugar, pueden agregarse permisos de ejecución para ejecutar el archivo como si se tratara de cualquier ejecutable (gracias al *shebang*):

```
$ chmod u+x primerScript.sh
$ ./primerScript.sh
Hola, mundo
```

Variables

El lenguaje de la *shell* es un lenguaje de programación completo, y también en este lenguaje existe el concepto de variables, estructuras de control, funciones, entre otros.

Una variable es definida al momento de inicializarse, y para hacer referencia a la misma se utiliza el símbolo $

```
#!/bin/bash
MENSAJE="Hola, mundo"
echo $MENSAJE
```

Leyendo entrada del usuario

Con `read` pueden leer una línea desde la entrada estándar

```
#!/bin/bash
read entrada
echo $entrada
```

Variables argumentales

En un *script de shell* hay variables especiales para ver los argumentos que fueron recibidos. Estas llevan como nombre un número entero no negativo ($0 guarda el nombre del archivo del *script*):

```
#!/bin/bash
echo "Mi primer argumento es: $1"
echo "Mi segundo argumento es: $2"
```

Y entonces:

```
$ ./argumentos.sh Hola mundo
Mi primer argumento es: Hola
Mi segundo argumento es: mundo
```

Variables argumentales

La variable $# indica el número de argumentos que recibió el *script*, mientras que $@ muestra todos los argumentos separados por espacios:

```
#!/bin/bash
echo "Mi primer argumento es: $1"
echo "Mi segundo argumento es: $2"
echo "Recibí $# argumentos: $@ "
```

Y entonces:

```
$ ./argumentos.sh Hola mundo
Mi primer argumento es: Hola
Mi segundo argumento es: mundo
Recibí 2 argumentos: Hola mundo
```

Conociendo el código de salida de un archivo

¿Recuerdas el `return 0;` de tu función `main` en lenguaje C? Este es el código de salida del programa. Un `0` indica que no hubo errores, cualquier otro valor significa que algo salió mal. La variable `$?` guarda el código de salida del último comando:

```
$ ls archivo-que-no-existe
ls: cannot access blah blah...
$ echo "$?"
2
```

Sustitución de comandos

Con la sintaxis "$(comando)" puedes conseguir que la cadena encerrada entre comillas se convierta en la salida estándar de comando:

```
#/bin/bash

echo "Esta es la información de tu usuario"
id $(whoami)

kernel="$(uname -s)"

echo "Estás usando $kernel"
```

```
Esta es la infomación de tu usuario juanito:
uid=1000(juanito) gid=100(users) ...
Estás usando Linux
```

Imprimiendo muchas cosas

Si se necesita imprimir una sección larga de texto, en lugar de tener muchos comandos `echo`, se puede utilizar el concepto de `here document`:

```
#!/bin/sh
FECHA=$(date)

cat <<EOF
Fecha: $FECHA

Voy a imprimir muchas cosas

Dolor illo autem excepturi in lorem culpa. Suscipit ut sed
...
recusandae non quia provident assumenda Delectus beatae an
EOF
```

Condicionales

Los condicionales utilizan el código de salida de algún otro comando.
0 indica verdadero mientras que cualquier otro valor indica falso.

```
#!/bin/bash

if [ "$1" = "Hola" ]; then
    echo 'Recibí "Hola" como argumento'
fi
```

Condicionales

```
#!/bin/bash

if [ "$1" = "Hola" ]; then
    echo 'Recibí "Hola" como argumento'
elif [ "$1" = "Bye" ]; then
    echo 'Recibí "Bue" como argumento'
else
    echo 'Recibí otra cosa'
fi
```

Condicionales

Además de utilizar [] en condicionales, se puede escribir directamente un comando y utilizar su código de salida para determinar si la condición es cierta o no:

```
#!/bin/bash

if grep -q juanito /etc/passwd; then
    echo "juanito está en el archivo passwd"
else
    echo "juanito no está en el archivo passwd..."
fi
```

Operadores lógicos

Tenemos los operadores `&& || y !`.

```
#!/bin/bash

# Cuidado con los espacios al interior de []!!
# Si estás dentro de [], puede usar
# -o y -a en lugar de || y &&
if [ "$1" = "hola" ] || [ "$2" = "adios" ]; then
# if [ "$1" = "hola" -o "$2" = "adios" ]; then
    echo 'Recibí "hola" o "adios"'
fi

if [ ! "$3" = "mundo" ]; then
    echo 'El tercer argumento no es "mundo"'
fi
```

Operadores lógicos

Estos operadores son *lazy*, así que en esta expresión:

`comando1 && comando2`

`comando2` se ejecutará sólo si `comando2` terminó con código 0.

De igual forma, en:

`comando1 || comando2`

`comando2` se ejecutará sólo si `comando2` terminó con código distinto a 0.

Chequeos

Hay ciertos operadores unarios que funcionan dentro de `[]`:

Operador	Checa
`-f`	Es un archivo normal
`-d`	Es directorio
`-h`	Es un enlace simbólico
`-b`	Es un archivo de bloque
`-c`	Es un archivo de caracteres
`-e`	El archivo existe
`-s`	El archivo no está vacío

Chequeos de permisos

Operador	Permiso
`-r`	Lectura
`-w`	Escritura
`-x`	Ejecución

Chequeos aritméticos

Operador	El primer elementos es _____ que el segundo
`-ne`	no igual
`-lt`	menor
`-gt`	mayor
`-le`	menor o igual
`-ge`	mayor o igual

Case

Es similar al switch en otros lenguajes:

```
case $1 in
    adios)
        echo "Hasta luego!"
        ;;
    hola|saludos)
        echo "Hola!"
        ;;
    tecuento*)
        echo "Cuéntame más"
        ;;
    *)
        echo "¿Cómo?"
        ;;
esac
```

Ciclos

El ciclo `for` es como un ciclo `for each` en otros lenguajes:

```
for nombre in pedro juan alberto; do
    echo $nombre
done
```

Puedes utilizar el programa `seq` para crear una secuencia de elementos:

```
for i in $(seq 10); do
    echo $i
done
```

Ciclos while

También existen los ciclos while. Este ejemplo lee un archivo que recibe como argumento línea por línea y muestra sun contenidos:

```
#!/usr/bin/bash

while read -r line;
do
    echo $line
done < "$1"
```