Actividad 4

Fernando Leyva Cárdenas

March 3, 2018

1 Introducción

En esta actividad nos apoyamos con un intérprete de comandos (Shell), quien es juega el papel de intermediario entre el usuario y el sistema operativo. Entre ellos están: C Shell (/bin/csh), Bourne Shell (/bin/sh), Korn Shell (/bin/ksh/), Bourne Again Shell (/bin/bash), y otros. En el caso de la mayoría de las variantes de Linux y macOS, por default viene el intérprete /bin/bash.

2 Comandos (Unix)

Basicamente lo que se hizo en esta actividad fue probar e identificar los comandos, los que usamos son los siguentes:

- cat
- chmod
- echo
- grep
- less
- ls
- wc
- Redirectores |, >

¿Entoces para que sirven estos comandos?, daremos una simple descripcion de cada uno de ellos:

1. El **comando cat** es una utilidad estándar de Unix que lee archivos secuencialmente y los escribe en una salida estándar. El nombre se deriva de su función para concatenar archivos.

   La Especificación Unix define la operación de cat para leer archivos en la secuencia dada en sus argumentos, escribiendo sus contenidos en el resultado estándar en la misma secuencia. La especificación exige el soporte de una bandera de opción, u para salida sin búfer, lo que significa que cada byte se escribe después de haber sido leído. Algunos sistemas operativos, como los que usan GNU Core Utilities, lo hacen de forma predeterminada e ignoran el indicador.

   Si uno de los nombres de los archivos de entrada se especifica como un solo guión (-), entonces cat lee de la entrada estándar en ese punto de la secuencia. Si no se especifican archivos, cat lee solo desde la entrada estándar.

2. En sistemas operativos tipo Unix, **chmod** es el comando y la llamada al sistema que pueden cambiar los permisos de acceso a los objetos del sistema de archivos (archivos y directorios). También puede alterar los indicadores de modo especial. La solicitud es filtrada por umask. El nombre es una abreviatura del modo de cambio.

3. **echo** es un comando para la impresión de un texto en pantalla. Es utilizado en las terminales de los sistemas operativos como Unix, GNU/Linux, o MS-DOS; dentro de pequeños programas llamados scripts; y en ciertos lenguajes de programación tales como PHP. En Unix (y otros sistemas derivados), el comando echo permite utilizar variables y otros elementos del intérprete de comandos. En las implementaciones más comunes y usadas como Bash, echo se trata de una función built-in, es decir, una función interna del intérprete de comandos y no un programa externo, así como cat o grep.

4. **grep** es una utilidad de la línea de comandos escrita originalmente para ser usada con el sistema operativo Unix.

   Usualmente, grep toma una expresión regular de la línea de comandos, lee la entrada estándar o una lista de archivos, e imprime las líneas que contengan coincidencias para la expresión regular.

   Su nombre deriva de un comando en el editor de texto ed que tiene la siguiente forma: g/re/p y significa «hacer una búsqueda global para las líneas que encajen con la expresión regular (regular expression en inglés), e imprimirlas». Hay varios argumentos que se pueden usar con grep para modificar el comportamiento por defecto.

   rep generalmente ejecuta alguna variante del algoritmo Boyer-Moore (para búsqueda de strings), utilizando expresiones regulares para definir la consulta. Puede manejar archivos, directorios (y subdirectorios), o la entrada estándar (stdin).

   El programa es configurable por medio de opciones de invocación, pudiendo (por ejemplo) mostrar las líneas con aciertos, desaciertos, el contexto de la coincidencia, etc.

5. El comando **less** es utilizado para ver (pero no cambiar) el contenido de un archivo de texto una pantalla a la vez; tiene la capacidad extendida de permitir tanto la navegación hacia adelante como hacia atrás a través del archivo. A diferencia de la mayoría de los editores de texto de Unix / viewers, less no necesita leer todo el archivo antes de comenzar, lo que resulta en tiempos de carga más rápidos con archivos de gran tamaño. se puede invocar con opciones para cambiar su comportamiento, por ejemplo, la cantidad de líneas que se mostrarán en la pantalla. Algunas opciones varían según el sistema operativo.

   Por defecto, less muestra el contenido del archivo a la salida estándar (una pantalla a la vez). Si se omite el argumento de nombre de archivo, muestra los contenidos de la entrada estándar (generalmente, la salida de otro comando a través de un conducto). Si la salida se redirige a otra cosa que no sea un terminal , por ejemplo, un conducto a otro comando, menos se comporta como **cat** .

6. **ls** es un comando del sistema operativo Unix y derivados que muestra un listado con los archivos y directorios de un determinado directorio. Los resultados se muestran ordenados alfabéticamente.

   Los archivos y directorios cuyo nombre comienza con . (punto) no se muestran con la instrucción ls, por lo que se suelen denominar «archivos ocultos». La opción -a de ls inhibe este comportamiento, y muestra todos los archivos y subdirectorios, incluso los que comienzan con punto.

7. El comando **wc** es un comando utilizado en el sistema operativo Unix que permite realizar diferentes conteos desde la entrada estándar, ya sea de palabras, caracteres o saltos de líneas.

   El programa lee la entrada estándar o una lista concatenada y genera una o más de las estadísticas siguientes: conteo de líneas, conteo de palabras, y conteo de bytes. Si se le pasa como parámetro una lista de archivos, muestra estadísticas de cada archivo individual y luego las estadísticas generales.

Por lo tanto en resumen, se utilizaron estos comando en esta actividad 4 para descargar 12 archivos de cierta parte del mundo, que al utlizar los comandos los pasamos a otro archivo, los comparamos y jugamos un poco con ellos, como seria el caso cuando asignabamos los permisos de quien lo puede leer, modificar y/o ejecutar, esto con chmod.

3 Sintesis (Shell Script Tutorial)

La programación de script de Shell tiene una mala impresión entre algunos administradores de sistemas de Unix. Esto es normalmente debido a una de dos cosas: La velocidad a la que se ejecutará un programa interpretado en comparación con un programa C, o incluso un programa Perl interpretado. Dado que es fácil escribir un script de shell de tipo de trabajo por lotes simple, hay muchos scripts de shell de mala calidad.

Hay una serie de factores que pueden incluirse en scripts de shell buenos, limpios y rápidos. Los criterios más importantes deben ser un diseño claro y legible; La segunda es evitar comandos innecesarios.

Una debilidad en muchos scripts de shell es líneas tales como:

```
cat / tmp / myfile | grep "mystring"
```

que se ejecutará mucho más rápido como:

```
grep "mystring" / tmp / myfile
```

Por supuesto, este tipo de cosas es lo que el sistema operativo está ahí, y normalmente es bastante eficiente para hacerlo. Pero si este comando se ejecutara en un bucle varias veces, el guardado de no localizar y cargar el cat ejecutable, configurar y liberar el conducto, puede hacer una diferencia, especialmente en, por ejemplo, un entorno CGI donde hay suficientes otros factores ralentizar las cosas sin que el guión en sí sea un obstáculo demasiado grande. Algunos Unices son más eficientes que otros en lo que ellos denominan "crear y destruir procesos", es decir, cargarlos, ejecutarlos y eliminarlos de nuevo.

Casi todos los lenguajes de programación existentes tienen el concepto de variables , un nombre simbólico para un trozo de memoria al que podemos asignar valores, leer y manipular sus contenidos. Podemos establecer de manera interactiva nombres de variables usando el readcomando; la siguiente secuencia de comandos le pide su nombre y luego lo saluda personalmente, por ejemplo:

```
#! / bin / bash echo ¿Cuál es tu nombre ? lee el eco de MY_NAME "Hello $ MY_NAME - Espero que estés bien".
```

Esto está usando el comando shell-builtin read que lee una línea de entrada estándar en la variable suministrada. Tenga en cuenta que incluso si le da su nombre completo y no utiliza comillas dobles alrededor del comando echo, sigue emitiendo correctamente.

No es necesario declarar las variables en el shell Bourne, como sí lo hacen en lenguajes como C. Pero si intenta leer una variable no declarada, el resultado es la cadena vacía. No obtienes advertencias o errores. Esto puede causar algunos errores sutiles - si asigna:

```
MY_OBFUSCATED_VARIABLE=Hello
```

Y luego:

```
echo $MY_OSFUCATED_VARIABLE
```

Entonces no obtendrá nada (ya que el segundo OBFUSCATED está mal escrito).

Hay un comando llamado **export** que tiene un efecto fundamental en el alcance de las variables. Para saber realmente qué está pasando con sus variables, deberá comprender algo sobre cómo se usa. En el ejemplo hicimos:

```
#! / bin / sh echo "MYVAR es: $ MYVAR" MYVAR = "hola" eco "MYVAR es: $ MYVAR"
```

MYVAR no se ha establecido en ningún valor, por lo que está en blanco. Entonces le damos un valor, y tiene el resultado esperado.

Ahora hablaremos de lo que son los **comodines**, lo cual no es nada nuevo si se ha usado anteriormente unix. Esta sección es realmente solo para hacer que las viejas celdas grises piensen cómo se ven las cosas cuando estás en un guión de shell, prediciendo cuál es el efecto de usar diferentes sintaxis.

Ciertos personajes son importantes para el caparazón; hemos visto, por ejemplo, que el uso de caracteres de comillas dobles (") afecta la forma en que se tratan los espacios y los caracteres TAB, por ejemplo:

```
$ echo Hello World Hello World
$ echo "Hello World" Hello World
```

Entonces, ¿cómo mostramos Hello "World":?

```
$ echo "Hello \" World \ ""
```

Pero sin emgargo el siguente codigo:

```
$ echo "Hello" World ""
```

Se interpretaría como tres parámetros: "Hola ", Mundo , ""
Entonces la salida sería:

```
Hola mundo
```

Hay que tener en cuenta que perdemos las comillas por completo. Esto se debe a que las primeras y segundas comillas marcan los espacios Hello y following; el segundo argumento es un "Mundo" sin comillas y el tercer argumento es la cadena vacía; "".

La mayoría de los caracteres (*, ', etc) no se interpretan (es decir, que se toman literalmente) por medio de la colocación entre comillas dobles (""). Se toman como están y se pasan al comando que se llama. Un ejemplo usando el asterisco (*) :

```
$ echo * caso . shtml escapar . shtml primero . shtml

funciones . shtml consejos . índice shtml . shtml
ip - primer . txt raid1 + 0.txt
$ echo * txt
ip - primer . txt raid1 + 0.txt
$ echo "*" *
$ echo "* txt" * txt
```

En el primer ejemplo, * se expande para indicar todos los archivos en el directorio actual, en el segundo ejemplo, * txt significa todos los archivos que terminan en txt, en el tercero, ponemos * entre comillas dobles, y se interpreta literalmente, en el cuarto ejemplo, lo mismo se aplica, pero hemos agregado txt a la cadena. Sin embargo, los simbolos todavía son interpretados por el shell, incluso cuando están entre comillas dobles. El carácter de barra invertida se utiliza para marcar estos caracteres especiales para que el intérprete no los interprete, sino que los pase al comando que se está ejecutando (por ejemplo, echo).

La mayoría de los lenguajes tienen el concepto de bucles: si queremos repetir una tarea veinte veces, no queremos tener que escribir el código veinte veces, con un ligero cambio cada vez. Como resultado, tenemos bucles fory while en el shell Bourne.

Para **for** los bucles iteran a través de un conjunto de valores hasta que se agote la lista:

```
    #!/bin/sh
for i in 1 2 3 4 5
do
  echo "Looping ... number $i"
done
```

Y obtenemos como resultado:

```
Looping .... number 1
Looping .... number 2
Looping .... number 3
Looping .... number 4
Looping .... number 5
```

En **while**:

```
#! / bin / bash
INPUT_STRING = hola
 mientras [ "$ INPUT_STRING" ! = "bye" ] no
  echo "Por favor escriba algo en (adiós a dejar de fumar)"

  leer INPUT_STRING
  echo "Escribiste: $ INPUT_STRING" hecho
```

Lo que ocurre aquí es que las instrucciones de eco y lectura se ejecutarán indefinidamente hasta que escriba "bye" cuando se le solicite.

Otro concepto importante es la **prueba**, es utilizada por prácticamente todos los guiones de shell escritos. Puede que no parezca así, porque a menudo test no se llama directamente. test es más frecuentemente llamado as . Es un enlace simbólico test, solo para hacer que los programas de shell sean más legibles. También es normalmente un shell incorporado (lo que significa que el intérprete de comandos interpretará el significado test, incluso si su entorno Unix está configurado de manera diferente.

La prueba a menudo se invoca indirectamente a través de las instrucciones if y while. También es la razón por la que tendrás problemas si creas un programa llamado test y tratas de ejecutarlo, ya que se llamará a este intérprete de comandos en lugar de tu programa; La sintaxis para if...then...else...es:

```
if [ ... ]
then
  # if-code
else
  # else-code
fi
```

El caso es una declaración guarda al pasar por un conjunto completo de if .. then .. else de declaraciones. Su sintaxis es realmente bastante simple:

```
#!/bin/bash

echo "Please talk to me ..."
while :
do
  read INPUT_STRING
  case $INPUT_STRING in
hello)
echo "Hello yourself!"
;;
bye)
echo "See you again!"
break
;;
*)
echo "Sorry, I don't understand"
;;
  esac
done
echo
echo "That's all folks!"
```

La sintaxis es bastante simple: la linea del caso en sí tiene siempre el mismo formato, y significa que estamos probando el valor de la variable INPUT STRING.

Ya hay un conjunto de variables establecidas para nosotros, y la mayoría de ellas no pueden tener valores asignados. Estos pueden contener información útil, que el script puede utilizar para conocer el entorno en el que se está ejecutando.

El primer conjunto de variables que veremos son $0 .. $9 y $#. La variable $0 es el nombre base del programa como se lo llamó. $1 .. $9son los primeros 9 parámetros adicionales con los que se invocó el script. La variable $ es todos los parámetros $1 .. whatever. La variable $*, es similar, pero no conserva ningún espacio en blanco, y las comillas, por lo que "Archivo con espacios" se convierte en "Archivo" "con" "espacios". Esto es similar a echolo que vimos en A First Script . Como regla general, usa $@ y evita $*. $# es la cantidad de parámetros con los que se invocó el script. Por ejemplo:

```
#!/bin/bash
echo "I was called with $# parameters"
echo "My name is $0"
echo "My first parameter is $1"
echo "My second parameter is $2"
echo "All parameters are $@"
```

Las otras dos variables principales que le asigna el entorno son $$ y $!. Estos son ambos números de proceso. La variable $$ es el PID (identificador de proceso) del shell actualmente en ejecución. Esto puede ser útil para crear archivos temporales, como la /tmp/my-script.$$ la cual es útil si se pueden ejecutar muchas instancias del script al mismo tiempo, y todas necesitan sus propios archivos temporales.

La variable $! es el PID del último proceso de fondo de ejecución. Esto es útil para realizar un seguimiento del proceso a medida que avanza con su trabajo.

Otra variable interesante es IFS. Este es el separador de campo interno . El valor predeterminado es SPACE TAB NEWLINE, pero si lo está cambiando, es más fácil tomar una copia, por ejemplo:

```
#!/bin/sh
old_IFS="$IFS"
IFS=:
echo "Please input some data separated by colons ..."
read x y z
IFS=$old_IFS
echo "x is $x y is $y z is $z"
```

Como mencionamos , las llaves alrededor de una variable evitan confusiones, Sin embargo, eso no es todo, estos brackets de lujo tienen otro uso mucho más poderoso. Podemos tratar con problemas de variables indefinidas o nulas (en el shell, no hay mucha diferencia entre indefinido y nulo).

Consideremos el siguiente fragmento de código que solicita al usuario la entrada, pero acepta los valores predeterminados:

```
#!/bin/sh
echo -en "What is your name [ `whoami` ] "
read myname
if [ -z "$myname" ]; then
  myname=`whoami`
fi
echo "Your name is : $myname"
```

Pasar el " -en" a echo le dice que no agregue un salto de línea (para bash y csh). Para Dash, Bourne y otros shells compatibles, en su lugar, utiliza un " " al final de la línea. Ksh entiende ambas formas; Esta secuencia de comandos se ejecuta así si acepta el valor predeterminado presionando "RETORNO":

```
steve$ ./name.sh
What is your name [ steve ]
Your name is : steve
```

Por otro lado, hablaremos de los programas externos, Los programas externos a menudo se usan en scripts de shell; hay algunas órdenes internas (echo, which, y test son comúnmente incorporados), pero muchos comandos útiles son en realidad utilidades Unix, tales como tr, grep, expry cut; El backtick (‘) también se asocia a menudo con comandos externos. Debido a esto, discutiremos primero el contragolpe; El backtick se usa para indicar que el texto adjunto se debe ejecutar como un comando. Esto es bastante simple de entender. Primero, use un intérprete interactivo para leer su nombre completo desde /etc/passwd:

```
$ grep "^${USER}:" /etc/passwd | cut -d: -f5
Steve Parker
```

Ahora tomaremos esta salida en una variable que podemos manipular más fácilmente:

```
$ MYNAME=`grep "^${USER}:" /etc/passwd | cut -d: -f5`
$ echo $MYNAME
Steve Parker
```

Entonces vemos que el backtick simplemente captura el resultado estándar de cualquier comando o conjunto de comandos que decidamos ejecutar. También puede mejorar el rendimiento si desea ejecutar un comando lento o un conjunto de comandos y analizar varios bits de su salida:

```
#!/bin/sh
find / -name "*.html" -print | grep "/index.html$"
find / -name "*.html" -print | grep "/contents.html$"
```

Una característica que a menudo se pasa por alto de la programación de guiones de shell de Bourne es que puede escribir fácilmente funciones para usar en su secuencia de comandos. Esto generalmente se hace de una de dos maneras; con un script simple, la función simplemente se declara en el mismo archivo como se llame. Sin embargo, al escribir un conjunto de secuencias de comandos, a menudo es más fácil escribir una ”biblioteca” de funciones útiles, y el origen de ese archivo al inicio de los otros scripts que utilizan las funciones.

Podría haber cierta confusión acerca de si llamar a funciones o procedimientos de shell funciones ; la definición de una función es tradicionalmente que devuelve un solo valor y no genera nada. Un procedimiento, por otro lado, no devuelve un valor, pero puede producir un resultado. Una función de shell no puede hacer ni una ni la otra ni ambas. En general, se acepta que en los scripts de shell se les llama funciones; Una función puede devolver un valor en una de cuatro formas diferentes:

- Cambiar el estado de una variable o variables
- Use el comando exit para finalizar el script de shell
- Utilice el comando return para finalizar la función y devolver el valor proporcionado a la sección de llamada del script de shell
- cho output to stdout, que será capturado por la persona que llama al igual que c = ‘expr $ a + $ b‘ está atrapado

Esto es como C, que exit detiene el programa y return devuelve el control a la persona que llama, la diferencia es que una función de shell no puede cambiar sus parámetros, aunque puede cambiar los parámetros globales.

Los programadores acostumbrados a otros lenguajes pueden sorprenderse con las reglas de alcance para las funciones de shell. Básicamente, no hay una definición del alcance, aparte de los parámetros ($1, $2, $@, etc); tomando como ejemplo el siguiente codigo:

```
#!/bin/sh

myfunc()
{
  echo "I was called as : $@"
```

```
  x=2
}

### Main script starts here

echo "Script was called with $@"
x=1
echo "x is $x"
myfunc 1 2 3
echo "x is $x"
```

El script, cuando se llama as scope.sh a b c, da el siguiente resultado:

```
Script was called with a b c
x is 1
I was called as : 1 2 3
x is 2
```

Los parametros $@ se cambian dentro de la función para reflejar cómo se llamó a la función. x Sin embargo, la variable es efectivamente una variable global, la myfunc cambió y ese cambio sigue siendo efectivo cuando el control vuelve al guión principal; Las funciones tampoco pueden cambiar los valores con los que han sido llamados; esto debe hacerse cambiando las variables en sí mismas, no los parámetros pasados al script.

Las funciones pueden ser recursivas: aquí hay un ejemplo simple de una función factorial:

```
#!/bin/sh

factorial()
{
  if [ "$1" -gt "1" ]; then
    i=`expr $1 - 1`
    j=`factorial $i`
    k=`expr $1 \* $j`
    echo $k
  else
    echo 1
  fi
}


while :
do
  echo "Enter a number:"
  read x
  factorial $x
done
```

nix está lleno de utilidades de manipulación de texto, algunas de las más poderosas; Prácticamente todo lo que puede pensar está controlado por un archivo de texto o por una interfaz de línea de comandos (CLI). Lo único que no puede automatizar con un script de shell es una utilidad o función solo GUI. ¡Y en Unix, no hay muchos de ellos!.

Bash tiene algunas herramientas de búsqueda de historia muy prácticas; las teclas de flecha hacia arriba y hacia abajo se desplazarán por el historial de comandos anteriores. Más útilmente, Ctrl + r hará una búsqueda inversa, haciendo coincidir cualquier parte de la línea de comando. Presione ESC y el comando seleccionado se pegará en el shell actual para que pueda editarlo según sea necesario.

Si desea repetir un comando que ejecutó anteriormente y sabe con qué caracteres comenzó, puede hacer esto:

```
bash$ ls /tmp
(list of files in /tmp)
bash$ touch /tmp/foo
bash$ !l
ls /tmp
(list of files in /tmp, now including /tmp/foo)
```

Puede hacer que ksh sea más útil agregando comandos de historial, ya sea en vio emacsmodo. Hay varias maneras de hacerlo, dependiendo de las circunstancias exactas. set -o vi„ ksh -o vio exec ksh -o vi(donde "vi" podría reemplazarse por "emacs" si prefiere el modo emacs).

4 Apendice

1. **¿Qué fue lo que más te llamó la atención en esta actividad?** Los comandos para crear scripts
2. **¿Qué consideras que aprendiste?** como utilizar los comando para hacer diversas tarea desde la terminal
3. **¿Cuáles fueron las cosas que más se te dificultaron?** el formato y la secuencia en que se debian ejecutar
4. **¿Cómo se podría mejorar en esta actividad?** tal vez con videos tutoriales
5. **¿En general, cómo te sentiste al realizar en esta actividad?** sientp que aprendi mucho

5 Bibliografia

- wikipedia, cat (unix) ,$URL : https : //en.wikipedia.org/wiki/Cat_(Unix)$, 03/03/2018
- wikipedia, chmod (unix), $URL : https : //en.wikipedia.org/wiki/Chmod$, 03/03/2018
- wikipedia, echo (informatica), $URL : https : //es.wikipedia.org/wiki/Echo_(inform\%C3\%A1tica)$, 03/03/2018
- howtoforge, grep (unix), URL: $https : //www.howtoforge.com/tutorial/linux - grep - command/$, 03/03/2018
- wikipedia, less (unix), URL: $https : //en.wikipedia.org/wiki/Less_(Unix)$, 03/03/2018
- wikipedia, ls, URL: $https : //es.wikipedia.org/wiki/Ls$, 03/03/2018
- wikipedia, wc (unix), URL: $https : //es.wikipedia.org/wiki/Wc_(Unix)$, 03/03/2018
- Steve Parker, Shell Script Tutorial, URL: $https : //www.shellscript.sh/index.html$, 03/03/2018