HFC

INTRODUCIÓN A UNIX

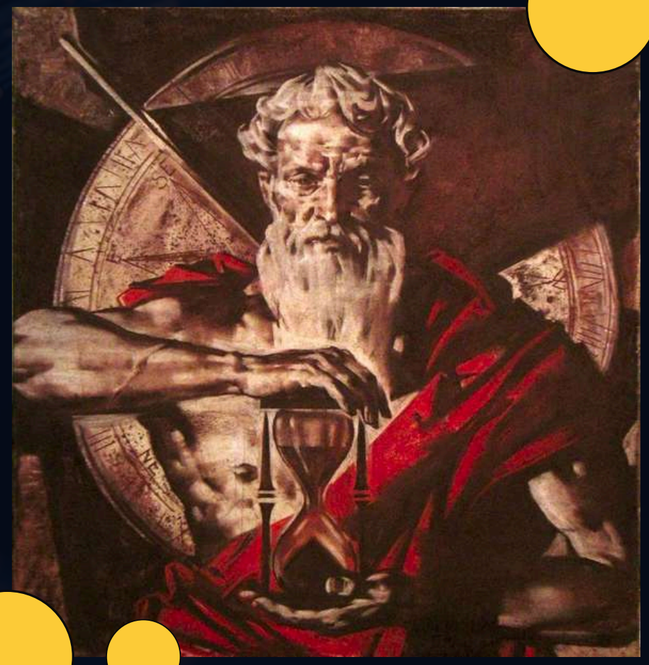
DANNA MÁRQUEZ
FERNANDO ROMERO

CRON JOBS

Son tareas programadas que se ejecutan automáticamente en momentos específicos. Son gestionados por el demonio cron y se configuran en el archivo crontab. Se encuentran en /etc/crontab.

En /etc/cron.d contiene archivos de configuración que permiten programar tareas en cron.

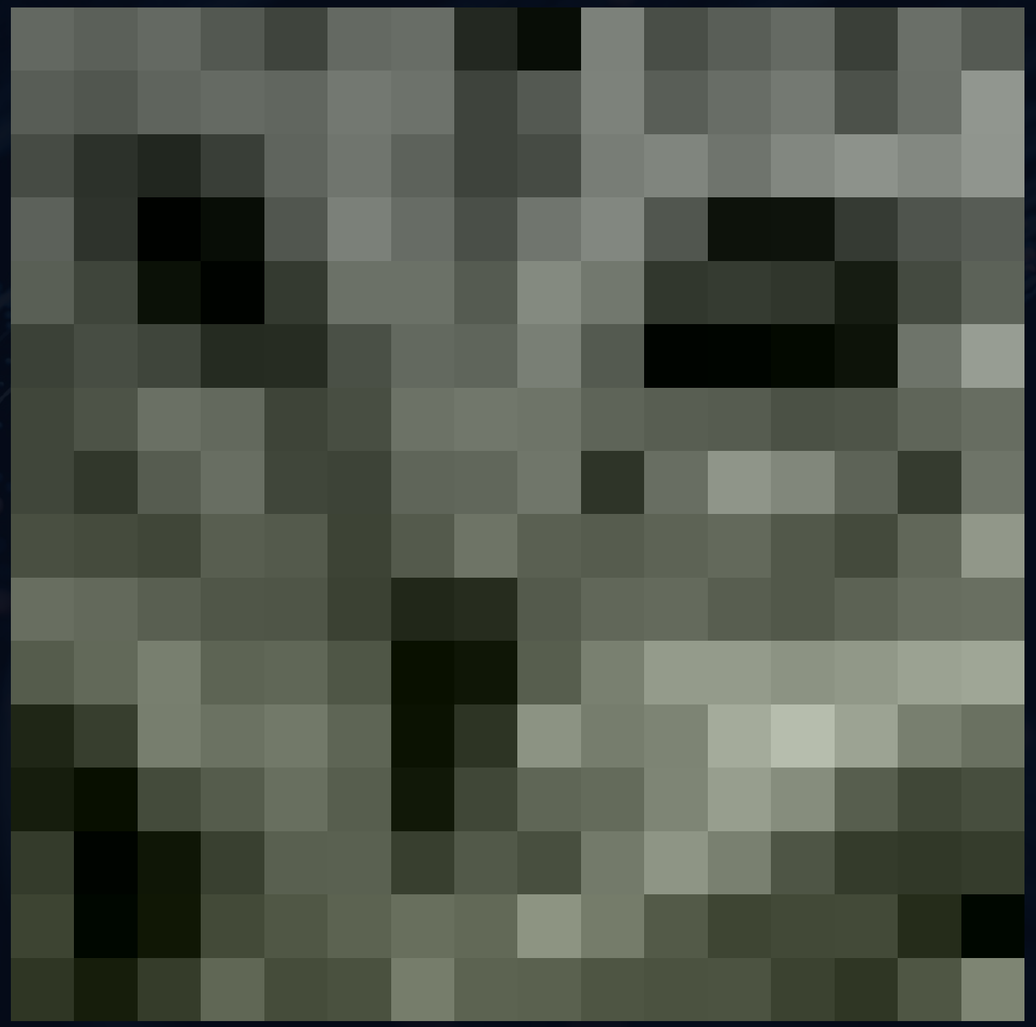
- Respaldos: Copiar archivos y bases de datos periódicamente.
- Actualizaciones: Aplicar parches o actualizar paquetes sin intervención manual.
- Monitoreo: Registrar métricas de uso del sistema o enviar alertas.

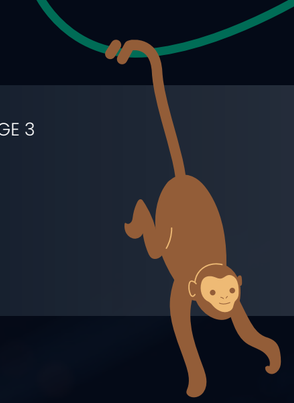


CRON JOBS

```
# * * * * *  command to execute
# ┬ ┬ ┬ ┬ ┬
# │ │ │ │ │
# │ │ │ │ │
# │ │ │ │ └──── day of week (0 - 7)
# │ │ │ └─────── month (1 - 12)
# │ │ └──────── day of month (1 - 31)
# │ └───────── hour (0 - 23)
# └──────── min (0 - 59)
```

OVER THE WIRE





CRONTAB

Comando	Descripción
`crontab -e`	Editar los cron jobs del usuario actual.
`crontab -l`	Listar los cron jobs del usuario actual.
`crontab -r`	Eliminar todos los cron jobs del usuario actual.
`crontab -i -r`	Pedir confirmación antes de eliminar los cron jobs.

BASH SCRIPTING

Conforme el **impacto** y la **utilidad** de las terminales aumentaron, la necesidad de automatizar procesos o secuencias de comando creció, surgiendo así el ***Scripting*** como una solución y planteando las bases para la simulación de *programación* a partir de instrucciones de terminal.

El **Bash Scripting** no es más que eso, definir una **lista de instrucciones** en un archivo que deseamos ejecutar y despues pasarselo al intérprete adecuado, convencionalmente **bash**, para que las **ejecute**.

Hacer Scripting **NO** es programar, debemos estar concientes de que seguimos ejecutando comandos en terminal, con las acciones que estos generan y con las salidas que estos imprimen en pantalla.

ACLARACIONES

Recordemos que es buena práctica siempre indicar un **shebang** al inicio del archivo, para ser claros con el intérprete que deseamos utilizar.

Este es el tipico “comentario” **#!/ruta/intérprete** que han visto en otros ejemplos de *Scripts* en *Python*.

La razón de que se llame **Bash Scripting** y no **Zsh Scripting** o cualquier otro *Shell* es que se enfoca la ejecución en este intérprete pues está instalado por defecto en todos los sistemas *Linux* y se considera el **intérprete** más estable, convencional y con configuraciones más regulares para realizar *Scripting*, independientemente del intérprete que utilicen en su día a día.

COMILLAS

En la terminal, podemos encerrar texto entre comillas simples o dobles para específicar que forman parte de un solo argumento, pero éstas realmente tienen funcionalidades distintas y distintas formas de especificarlas.

- **“”** : Se interpretan las variables, las **sustituciones de comandos** y los caracteres especiales **escapados** (\n, \t, \x61)
- **‘’** : Se interpreta **literalmente** todo lo que se encuentre dentro, no hay ningun tipo de escapes y no puede haber otra comilla (‘) en el texto.
- **$‘’** : Similar a encerrar con **‘** pero si se interpretan los escapes y nada más.

- **** : Define los escapes, que son caracteres especiales como saltos de línea (\n), tabulaciones (\t), etc. También define cuando se desea que se tome literalmente un caracter problemático, por ejemplo un ” adentro de comillas dobles (**“”**) sería: **“Mira una comilla doble \” y no hay problema”**

COMODINES

Existen comodines que podemos utilizar para referirnos a un conjunto de archivos que concuerden con el patrón indicado mediante estos **comodines**.

Estos archivos coincidentes remplazan el patrón indicado a la hora de ejecutar el comando

- ***** : Se refiere a cualesquiera 0 o más caracteres
- **?**: Se refiere únicamente a 1 caracter, que puede ser cualquiera
- **[abc]resto**: Se refiere a cualquier archivo que empiece por **a**, por **b**, o por **c** y continúe inmediatamente despues con la palabra **resto**. (aresto, bresto, cresto)
- **[a-z]resto**: Lo mismo pero que empiece con cualquier letra entre la **a** y la **z**.
- **[^abc]resto** : Se refiere a cualesquiera archivos que **NO** empiecen con **a**, **b** o **c**, y que continuen con resto

EXPANSIÓN DE LLAVES

La **expansión de llaves** es un mecanismo para la generación de varias cadenas con distintas modificaciones, se diferencia de los comodines que vimos anteriormente porque su objetivo principal **NO** es definir patrones para la busqueda de archivos, si no **generar múltiples patrones** con las modificaciones indicadas.

Sin embargo, también pueden utilizarse para definir patrones más específicos.

- echo **inicio{cadena1,cadena2}fin**
- ls **inicio{cadena1,cadena2}fin**

También pueden utilizarse para generar secuencias de números o letras, de manera similar a *Python*, por ejemplo:
echo **{inicio..fin..intervalo}** (Solo números)
echo **{letra..letra}** (letras)

NÚMEROS

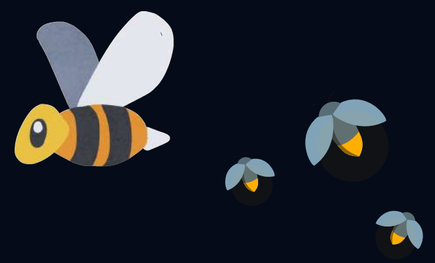
Para utilizar y procesar números desde la terminal, debemos encerrar los números y operaciones entre **$(( ))** y así obtener el **resultado** de estos en lugar de la cadena literal de la operación.

Los operandos son prácticamente iguales a los vistos en *Python*.

Operador	Descripción	Ejemplo
`+`	Suma	`echo $((5 + 3))` → `8`
`-`	Resta	`echo $((10 - 4))` → `6`
`*`	Multiplicación	`echo $((5 * 2))` → `10`
`/`	División entera	`echo $((9 / 2))` → `4`
`%`	Módulo (residuo)	`echo $((10 % 3))` → `1`
`**`	Exponente (no soportado, usa `bc`)	`echo "2^3"

NÚMEROS

Operador	Significado	Ejemplo
`-eq`	Igual a (`==`)	`[ 5 -eq 5 ]`
`-ne`	Distinto de (`!=`)	`[ 5 -ne 3 ]`
`-lt`	Menor que (`<`)	`[ 3 -lt 5 ]`
`-le`	Menor o igual que (`<=`)	`[ 3 -le 3 ]`
`-gt`	Mayor que (`>`)	`[ 5 -gt 3 ]`
`-ge`	Mayor o igual que (`>=`)	`[ 5 -ge 5 ]`



Uso de ((...))

Se usa para evaluar expresiones matemáticas sin necesidad de expr o bc

```
a=5
b=3
(( resultado = a + b ))
echo "Resultado: $resultado"  # Salida: 8
```

ESTRUCTURAS DE CONTROL

Con la necesidad de tomar el control del **flujo de ejecución** vino la definición de **comandos especiales** que conforman las estructuras de control más famosas utilizadas por lenguajes modernos.

- **if-else:** Ejecutar código basado en una condición
- **case:** Evaluar múltiples valores de una variable
- for: Iterar sobre listas o rangos
- **while**: Repetir mientras se cumpla una condición
- **until:** Repetir hasta que se cumpla una condición
- **break**: Salir de un bucle
- **continue**: Saltar una iteración de un bucle

IF

Estructura:

```
if [ condición ]
then
    # Código si la condición es verdadera
elif [ otra_condición ]; then
    # Código si la otra condición es verdadera
else
    # Código si ninguna condición se cumple
fi
```

Uso de [[...]]

Usada para evaluar condiciones.

```
nombre="Danna"
if [[ "$nombre" == "Danna" ]]; then
      echo "Hola, Danna!"
fi
```

FOR

Estructura:

```
for variable in lista; do
    # Comandos a ejecutar
done
```

Ejemplo:

```
for archivo in *.txt; do
    echo "Archivo encontrado: $archivo"
done
```

WHILE

Estructura:

```
while [ condición ]; do
    # Código a ejecutar mientras la condición sea verdadera
done
```

Ejemplo:

```
contador=1
while [ $contador -le 5 ]; do
    echo "Contador: $contador"
    ((contador++))  # Incrementar el contador
done
```

WHILE

Estructura:

```
while [ condición ]; do
    # Código a ejecutar mientras la condición sea verdadera
done
```

Ejemplo:

```
contador=1
while [ $contador -le 5 ]; do
    echo "Contador: $contador"
    ((contador++))  # Incrementar el contador
done
```

UNTIL

Estructura:

```
until [ condición ]; do
    # Código a ejecutar mientras la condición sea falsa
done
```

Ejemplo:

```
contador=1
until [ $contador -gt 5 ]; do
    echo "Contador: $contador"
    ((contador++))  # Incrementar el contador
done
```