Manual Completo de Bash: De Cero a Experto

Autor: Basdonax AI **Programa:** Fundamentos de Automatización y DevOps **Fecha:** Noviembre 2025

Índice

Nivel Básico: Primeros Pasos en la Terminal

Este nivel está diseñado para que te familiarices con el entorno de la terminal y los comandos fundamentales que te permitirán interactuar con el sistema operativo.

Capítulo 1: Introducción al Shell y Primeros Comandos

1.1. ¿Qué es Bash?

Introducción teórica: Bash (Bourne Again Shell) es el intérprete de comandos más popular en sistemas operativos basados en Unix, como Linux y macOS. Actúa como un puente entre el usuario y el kernel del sistema operativo, permitiendo ejecutar comandos, programas y scripts para automatizar tareas. Es el corazón de la administración de sistemas y la automatización en entornos de desarrollo y DevOps.

1.2. Comandos Básicos de Interacción

Tema: Mostrar mensajes y ubicación actual

Ejemplo práctico:

```
# 1. Mostrar un mensaje simple en la consola
echo "¡Hola, mundo desde Bash!"

# 2. Mostrar el nombre del usuario actual usando una variable de entorno
echo "El usuario actual es: $USER"

# 3. Mostrar la ruta absoluta del directorio de trabajo actual
pwd
```

Explicación línea por línea:

- `echo "¡Hola, mundo desde Bash!"`: El comando `echo` es uno de los más sencillos y se utiliza para imprimir texto o el valor de variables en la salida

estándar (la terminal).

- `echo "El usuario actual es: $USER"`: Aquí combinamos texto con una **variable de entorno** del sistema. `$USER` es una variable predefinida que almacena el nombre del usuario que está ejecutando la sesión.
- `pwd`: Significa "Print Working Directory" (Imprimir Directorio de Trabajo). Muestra la ruta completa (absoluta) del directorio en el que te encuentras actualmente.

Ejercicio propuesto: Crea un comando que muestre tu nombre de usuario, el nombre de tu máquina (variable `$HOSTNAME`) y la fecha actual (comando `date`).

Resultado esperado:

```
Mi usuario es: [tu_usuario]
Mi máquina se llama: [nombre_de_tu_máquina]
Hoy es: [fecha_y_hora_actual]
```

Solución:

```
echo "Mi usuario es: $USER" && echo "Mi máquina se llama: $HOSTNAME" && date
```

Capítulo 2: Manejo de Archivos y Directorios (Comandos Esenciales)

2.1. Navegación y Listado

Tema: Moverse y ver el contenido del sistema de archivos

Introducción teórica: El sistema de archivos en Unix/Linux está organizado de forma jerárquica, comenzando desde el directorio raíz (`/`). Los comandos de navegación te permiten moverte entre directorios y visualizar su contenido.

Ejemplo práctico:

```
# 1. Listar el contenido del directorio actual
ls

# 2. Listar el contenido con detalles (permisos, tamaño, fecha) y archivos
ocultos
ls -la

# 3. Moverse al directorio 'Documentos' (si existe en el directorio actual)
cd Documentos

# 4. Moverse al directorio padre (un nivel arriba)
cd ..

# 5. Moverse directamente al directorio personal (Home)
cd ~
```

Explicación línea por línea:

- `ls`: Lista los archivos y directorios visibles en el directorio actual.
- `ls -la`: Usa las opciones `-l` (formato largo) y `-a` (mostrar archivos ocultos, que empiezan con un punto).
- `cd Documentos`: El comando `cd` (Change Directory) te mueve a la ruta especificada. Si no se especifica una ruta absoluta, asume que es relativa al directorio actual.
- `cd ..`: El doble punto (`..`) es una referencia especial que siempre apunta al directorio inmediatamente superior (padre).
- `cd ~`: El tilde (`~`) es una referencia especial que siempre apunta a tu directorio personal (`/home/tu_usuario`).

Ejercicio propuesto: Desde tu directorio personal (`~`), navega a la carpeta `/tmp`, crea un archivo vacío llamado `prueba.txt` y luego regresa a tu directorio personal.

Resultado esperado: El archivo `prueba.txt` debe existir en `/tmp` y tu ubicación final debe ser `~`.

Solución:

```
cd /tmp
touch prueba.txt
cd ~
```

2.2. Creación, Copia y Eliminación

Tema: Manipulación básica de archivos y directorios

Introducción teórica: La manipulación de archivos es una tarea diaria en Bash. Los comandos `touch`, `mkdir`, `cp`, `mv` y `rm` son esenciales para crear, mover, copiar y eliminar elementos del sistema de archivos.

Ejemplo práctico:

```
# 1. Crear un nuevo directorio llamado 'mi_proyecto'
mkdir mi_proyecto

# 2. Crear un archivo vacío dentro del nuevo directorio
touch mi_proyecto/README.md

# 3. Copiar el archivo README.md a un nuevo archivo llamado LICENSE
cp mi_proyecto/README.md mi_proyecto/LICENSE

# 4. Mover (renombrar) el directorio 'mi_proyecto' a 'proyecto_final'
mv mi_proyecto proyecto_final

# 5. Eliminar el archivo LICENSE (¡Cuidado! No hay papelera de reciclaje)
rm proyecto_final/LICENSE

# 6. Eliminar el directorio y su contenido de forma recursiva y forzada
rm -rf proyecto_final
```

Explicación línea por línea:

- `mkdir mi_proyecto`: Crea un directorio.
- `touch mi_proyecto/README.md`: Crea un archivo vacío.
- `cp [origen] [destino]`: Copia el archivo de origen al destino.

- `mv [origen] [destino]`: Mueve un archivo o directorio. Si el destino es un nuevo nombre en la misma ubicación, funciona como un renombrado.
- `rm [archivo]`: Elimina un archivo.
- `rm -rf [directorio]`: **¡Comando peligroso!** Elimina directorios (`-r` de recursivo) y fuerza la eliminación sin preguntar (`-f` de forzado). Úsalo con extrema precaución.

Ejercicio propuesto: Crea una estructura de directorios `datos/2025/enero`, copia el archivo `/etc/hosts` a la carpeta `enero` y luego elimina la carpeta `datos` completa.

Resultado esperado: La carpeta `datos` y todo su contenido deben desaparecer.

Solución:

```
mkdir -p datos/2025/enero
cp /etc/hosts datos/2025/enero/
rm -rf datos
```

Resumen de Comandos y Conceptos Importantes (Nivel Básico)

Comando	Descripción	Opciones Comunes	Concepto Clave
`echo`	Imprime texto o variables en la terminal.	N/A	Salida Estándar
`pwd`	Muestra el directorio de trabajo actual.	N/A	Ruta Absoluta
`ls`	Lista el contenido de un directorio.	`-l` (detalles), `-a` (ocultos)	Sistema de Archivos
`cd`	Cambia el directorio de trabajo.	`..` (padre), `~` (home)	Navegación
`mkdir`	Crea un nuevo directorio.	`-p` (crea directorios padres si no existen)	Directorios
`touch`	Crea un archivo vacío o actualiza la marca de tiempo.	N/A	Archivos
`cp`	Copia archivos y directorios.	`-r` (recursivo para directorios)	Copia
`mv`	Mueve o renombra archivos y directorios.	N/A	Movimiento/Renombrado
`rm`	Elimina archivos.	`-r` (recursivo), `-f` (forzado)	Eliminación
`$USER`	Variable de entorno con el nombre de usuario.	N/A	Variables de Entorno

Ejercicio Integrador (Nivel Básico)

Objetivo: Crear una estructura de trabajo y un archivo de registro.

Instrucciones:

1. Crea un directorio llamado `curso_bash` en tu directorio personal.
2. Dentro de `curso_bash`, crea dos subdirectorios: `scripts` y `logs`.

3. Crea un archivo vacío llamado `registro.log` dentro del directorio `logs`.
4. Copia el archivo `registro.log` a la carpeta `scripts`, renombrándolo como `script_base.sh`.
5. Muestra la ruta absoluta de la carpeta `scripts`.
6. Muestra el contenido de la carpeta `curso_bash` (debe mostrar `logs` y `scripts`).

Solución (para verificar):

```
mkdir ~/curso_bash
mkdir ~/curso_bash/scripts ~/curso_bash/logs
touch ~/curso_bash/logs/registro.log
cp ~/curso_bash/logs/registro.log ~/curso_bash/scripts/script_base.sh
pwd ~/curso_bash/scripts
ls ~/curso_bash
```

Capítulo 3: Variables, Argumentos y Parámetros Especiales

3.1. Variables Personalizadas

Tema: Almacenar información para reutilizarla

Introducción teórica: Las variables son espacios de memoria que almacenan datos. En Bash, se crean simplemente asignando un valor a un nombre (sin espacios alrededor del signo `=`). Para acceder al valor, se antepone el signo de dólar (`$`).

Ejemplo práctico:

```bash
# 1. Definir una variable (sin espacios alrededor del =)
NOMBRE="Juan Pérez"

# 2. Definir una variable con un número
EDAD=30

# 3. Acceder y mostrar el valor de la variable
echo "Hola, mi nombre es $NOMBRE y tengo $EDAD años."

# 4. Usar llaves para delimitar la variable (buena práctica)
echo "El nombre es ${NOMBRE}."

# 5. Intentar usar la variable sin el signo $ (mostrará el texto literal)
echo "La variable es NOMBRE"
```

Explicación línea por línea:

- `NOMBRE="Juan Pérez"`: Asigna la cadena de texto "Juan Pérez" a la variable `NOMBRE`. Las comillas son opcionales si el valor no contiene espacios, pero son recomendadas.
- `EDAD=30`: Asigna el número 30 a la variable `EDAD`.
- `echo "Hola, mi nombre es $NOMBRE..."`: El signo `$` indica a Bash que debe sustituir el nombre de la variable por su valor.
- `echo "El nombre es ${NOMBRE}."`: El uso de llaves (`{}`) es útil para evitar ambigüedades, especialmente si la variable está seguida inmediatamente por texto.

Ejercicio propuesto: Define una variable llamada `RUTA_LOGS` con el valor `/var/log/`, y luego usa esa variable para listar el contenido de ese directorio.

Resultado esperado: El listado de archivos dentro de `/var/log/`.

Solución:

```bash
RUTA_LOGS="/var/log/"
ls $RUTA_LOGS
```

3.2. Parámetros Posicionales y Especiales

Tema: Recibir y gestionar argumentos en scripts

Introducción teórica: Cuando ejecutas un script, puedes pasarle información (argumentos). Bash asigna estos argumentos a variables especiales llamadas **parámetros posicionales** (`$1` , `$2` , etc.) y proporciona **parámetros especiales** para gestionar el script en sí.

Ejemplo práctico (Script: `parametros.sh`):

```
#!/bin/bash
# Este script muestra los argumentos pasados

echo "Nombre del script: $0"
echo "Primer argumento: $1"
echo "Segundo argumento: $2"
echo "Total de argumentos: $#"
echo "Todos los argumentos (como una cadena): $*"
echo "Todos los argumentos (como una lista): $@"
echo "Código de salida del último comando: $?"
```

Para ejecutar el script:

```
chmod +x parametros.sh
./parametros.sh archivo.txt 100
```

Explicación línea por línea:

- `$0` : Contiene el nombre del script que se está ejecutando (`./parametros.sh`).
- `$1` , `$2` : Contienen el primer y segundo argumento pasados al script (`archivo.txt` y `100`).
- `$#` : Contiene el número total de argumentos pasados (2).
- `$*` : Expande todos los argumentos como una única cadena de texto.
- `$@` : Expande todos los argumentos como elementos separados (útil en bucles).
- `$?` : Contiene el código de salida (status) del último comando ejecutado. `0` significa éxito.

Ejercicio propuesto: Crea un script llamado `saludo.sh` que reciba un nombre como primer argumento (`$1`) y muestre el mensaje: "Hola [Nombre], bienvenido al curso de Bash."

Resultado esperado (al ejecutar `./saludo.sh Pedro`):

```
Hola Pedro, bienvenido al curso de Bash.
```

Solución (Script `saludo.sh`):

```
#!/bin/bash
echo "Hola $1, bienvenido al curso de Bash."
```

Capítulo 4: Creación y Ejecución de Scripts

4.1. El Shebang y Permisos de Ejecución

Tema: Convertir comandos en un programa ejecutable

Introducción teórica: Un **script de Bash** es simplemente un archivo de texto que contiene una secuencia de comandos. Para que el sistema lo reconozca como un programa ejecutable, necesita dos cosas:

1. **El Shebang (`#!`):** La primera línea del script debe indicar qué intérprete debe usar el sistema para ejecutarlo. Para Bash, es casi siempre `#!/bin/bash`.
2. **Permisos de Ejecución:** El archivo debe tener el permiso de ejecución activado, lo cual se logra con el comando `chmod`.

Ejemplo práctico:

```
# 1. Crear un script simple (usando 'cat' para escribir rápidamente)
cat > mi_primer_script.sh << EOF
#!/bin/bash
# Script para mostrar la hora y el usuario
echo "La hora actual es: $(date)"
echo "El script fue ejecutado por: $USER"
EOF

# 2. Verificar los permisos actuales (debe mostrar '-rw-r--r--')
ls -l mi_primer_script.sh

# 3. Asignar permiso de ejecución al usuario (debe mostrar '-rwxr--r--')
chmod +x mi_primer_script.sh

# 4. Ejecutar el script
./mi_primer_script.sh
```

Explicación línea por línea:

- `cat > mi_primer_script.sh << EOF ... EOF`: Es una forma rápida de crear un archivo de texto con múltiples líneas (Heredoc).
- `#!/bin/bash`: El **Shebang**. Indica que el script debe ser ejecutado por el intérprete Bash.
- `$(date)`: Es una **sustitución de comando**. Ejecuta el comando `date` y sustituye esa parte de la línea por su salida.
- `ls -l`: Muestra los permisos del archivo. El primer carácter indica el tipo (`-` para archivo), y los siguientes nueve son los permisos (lectura, escritura, ejecución) para el usuario, el grupo y otros.
- `chmod +x`: Agrega el permiso de ejecución (`x`) al archivo.
- `./mi_primer_script.sh`: La forma estándar de ejecutar un script en el directorio actual. El `./` es necesario para indicar la ruta relativa.

Ejercicio propuesto: Crea un script llamado `limpieza.sh` que elimine todos los archivos con extensión `.tmp` en el directorio actual. Asegúrate de que sea ejecutable.

Resultado esperado: Al ejecutar el script, todos los archivos `.tmp` deben desaparecer.

Solución (Script `limpieza.sh`):

```
#!/bin/bash
# Script de limpieza
rm *.tmp
echo "Archivos .tmp eliminados."
```

Comandos de ejecución:

```
touch archivo1.tmp archivo2.txt archivo3.tmp
chmod +x limpieza.sh
./limpieza.sh
ls
```

Resumen de Comandos y Conceptos Importantes (Nivel Básico)

Comando/Concepto	Descripción	Ejemplo	Uso en Scripts
Variable	Almacena un valor.	`RUTA="/home/user"`	Reutilización de datos
`$`	Prefijo para acceder al valor de una variable.	`echo $RUTA`	Sustitución de valor
`$1`, `$2` ⋯	Parámetros posicionales (argumentos).	`./script.sh arg1` -> `$1` es `arg1`	Entrada de datos
`$#`	Número total de argumentos.	Si pasas 3 argumentos, `$#` es 3.	Validación de entrada
`$0`	Nombre del script.	`./mi_script.sh`	Mensajes de error
`$?`	Código de salida del último comando.	`0` (éxito), `>0` (error)	Control de errores
`#!/bin/bash`	**Shebang**. Indica el intérprete.	Primera línea del script.	Ejecución correcta
`chmod +x`	Otorga permiso de ejecución.	`chmod +x script.sh`	Hacer el script ejecutable
`./script.sh`	Ejecución de un script en el directorio actual.	`cd /tmp; ./script.sh`	Ejecución de programas

Ejercicio Integrador (Nivel Básico)

Objetivo: Crear un script de bienvenida personalizado.

Instrucciones:

1. Crea un script llamado `bienvenida.sh`.
2. El script debe recibir **dos argumentos**: el nombre de una persona (`$1`) y su edad (`$2`).

3. El script debe mostrar un mensaje que combine el nombre, la edad y la fecha actual.
4. Asegúrate de que el script sea ejecutable.

Resultado esperado (al ejecutar `./bienvenida.sh Ana 25`):

```
========================================
¡Bienvenida Ana!
Tienes 25 años.
Este mensaje fue generado el: [Fecha y Hora Actual]
========================================
```

Solución (Script `bienvenida.sh`):

```
#!/bin/bash
# Script de Bienvenida Personalizada

# Variables para mayor claridad
NOMBRE=$1
EDAD=$2
FECHA=$(date)

echo "========================================"
echo "¡Bienvenida $NOMBRE!"
echo "Tienes $EDAD años."
echo "Este mensaje fue generado el: $FECHA"
echo "========================================"
```

Comandos de ejecución:

```
chmod +x bienvenida.sh
./bienvenida.sh Ana 25
```

Nivel Intermedio: Control de Flujo y Procesamiento de Texto

Este nivel se enfoca en las estructuras de control que permiten a tus scripts tomar decisiones y repetir tareas, además de introducir herramientas poderosas para el procesamiento de datos y texto.

Capítulo 5: Condicionales y Estructuras de Control (`if`, `case`)

5.1. La Estructura `if-then-else`

Tema: Tomar decisiones basadas en condiciones

Introducción teórica: La estructura condicional `if` es fundamental en cualquier lenguaje de programación. Permite ejecutar un bloque de código solo si una condición es verdadera. En Bash, las condiciones se evalúan dentro de corchetes simples (`[ ]`) o dobles (`[[ ]]`), siendo los dobles más modernos y flexibles.

Ejemplo práctico:

```
#!/bin/bash
# Script para verificar si un número es mayor a 10

read -p "Ingresa un número: " NUMERO

if [ $NUMERO -gt 10 ]; then
    echo "El número $NUMERO es mayor que 10."
elif [ $NUMERO -eq 10 ]; then
    echo "El número es exactamente 10."
else
    echo "El número $NUMERO es menor que 10."
fi
```

Explicación línea por línea:

- `read -p "..." NUMERO`: Captura la entrada del usuario y la almacena en la variable `NUMERO`.
- `if [ $NUMERO -gt 10 ]; then`: Inicia la estructura `if`. La condición verifica si el valor de `$NUMERO` es **mayor que** (`-gt`) 10.
- `elif [ $NUMERO -eq 10 ]; then`: Es la abreviatura de "else if". Verifica si el número es **igual a** (`-eq`) 10.
- `else`: Se ejecuta si ninguna de las condiciones anteriores es verdadera.
- `fi`: Marca el final de la estructura `if`.

Operadores de Comparación Numérica Comunes:

Operador	Significado	Ejemplo
`-eq`	Igual a	`[ $A -eq $B ]`
`-ne`	No es igual a	`[ $A -ne $B ]`
`-gt`	Mayor que	`[ $A -gt $B ]`
`-ge`	Mayor o igual que	`[ $A -ge $B ]`
`-lt`	Menor que	`[ $A -lt $B ]`
`-le`	Menor o igual que	`[ $A -le $B ]`

Ejercicio propuesto: Crea un script que pida un nombre de archivo como argumento (`$1`) y use el condicional `if` para verificar si el archivo existe (`-f`) y si es legible (`-r`). Si ambas condiciones se cumplen, imprime "El archivo existe y es legible". Si no, imprime "El archivo no existe o no se puede leer".

Resultado esperado (al ejecutar con un archivo existente y legible):

```
El archivo existe y es legible
```

Solución (Fragmento de código):

```
if [ -f "$1" ] && [ -r "$1" ]; then
    echo "El archivo existe y es legible"
else
    echo "El archivo no existe o no se puede leer"
fi
```

5.2. La Estructura `case`

Tema: Manejar múltiples opciones de forma limpia

Introducción teórica: La estructura `case` es una alternativa más limpia y eficiente al uso de múltiples `elif` anidados, especialmente cuando se compara una única variable contra varios valores posibles (patrones).

Ejemplo práctico:

```
#!/bin/bash
# Script para manejar opciones de menú

read -p "Ingresa una opción (a, b, c): " OPCION

case $OPCION in
    a|A)
        echo "Seleccionaste la opción A. Iniciando proceso de respaldo."
        ;;
    b|B)
        echo "Seleccionaste la opción B. Ejecutando actualización de
sistema."
        ;;
    c|C)
        echo "Seleccionaste la opción C. Mostrando estado del servidor."
        ;;
    *)
        echo "Opción inválida. Por favor, elige a, b o c."
        ;;
esac
```

Explicación línea por línea:

- `case $OPCION in`: Inicia la estructura `case`, evaluando el valor de `$OPCION`.

- `a|A)`: Define un patrón. Si `$OPCION` es `a` **o** (`|`) `A`, se ejecuta el código siguiente.
- `echo "..."`: El código a ejecutar para ese patrón.
- `;;`: Marca el final del bloque de código para un patrón.
- `*)`: Es el patrón comodín, que coincide con cualquier valor que no haya coincidido antes (similar al `else` en un `if`).
- `esac`: Marca el final de la estructura `case`.

Ejercicio propuesto: Crea un script que reciba un argumento (`$1`) que sea un nombre de animal. Usa la estructura `case` para imprimir "Es un mamífero" si el animal es "perro" o "gato", y "Es un ave" si es "pájaro". Si no coincide, imprime "Animal desconocido".

Resultado esperado (al ejecutar `./animal.sh gato`):

```
Es un mamífero
```

Solución (Fragmento de código):

```
case $1 in
    perro|gato)
        echo "Es un mamífero"
        ;;
    pájaro)
        echo "Es un ave"
        ;;
    *)
        echo "Animal desconocido"
        ;;
esac
```

Capítulo 6: Bucles y Repetición de Tareas (`for`, `while`)

6.1. El Bucle `for`

Tema: Iterar sobre listas de elementos

Introducción teórica: El bucle `for` se utiliza para ejecutar un bloque de código repetidamente para cada elemento en una lista. Esta lista puede ser una secuencia de números, una lista de archivos, o cualquier conjunto de palabras.

Ejemplo práctico (Iteración sobre una lista de archivos):

```
#!/bin/bash
# Script para procesar archivos .log

echo "Iniciando procesamiento de logs..."

for ARCHIVO in *.log; do
    if [ -f "$ARCHIVO" ]; then
        echo "Procesando archivo: $ARCHIVO"
        # Aquí iría el código de procesamiento, por ejemplo, contar líneas
        LINEAS=$(wc -l < "$ARCHIVO")
        echo "  - Contiene $LINEAS líneas."
    fi
done

echo "Procesamiento finalizado."
```

Explicación línea por línea:

- `for ARCHIVO in *.log; do`: Inicia el bucle. La variable `ARCHIVO` tomará el valor de cada archivo que termine en `.log` en el directorio actual.
- `if [ -f "$ARCHIVO" ]; then`: Es una buena práctica verificar si el elemento es realmente un archivo antes de procesarlo.
- `wc -l < "$ARCHIVO"`: El comando `wc -l` cuenta las líneas. La redirección de entrada (`<`) pasa el contenido del archivo al comando.
- `done`: Marca el final del bucle `for`.

Ejercicio propuesto: Crea un bucle `for` que itere sobre los números del 1 al 5 e imprima: "Contando: [número]".

Resultado esperado:

```
Contando: 1
Contando: 2
Contando: 3
Contando: 4
Contando: 5
```

Solución (Fragmento de código):

```
for i in 1 2 3 4 5; do
    echo "Contando: $i"
done
# Alternativa con secuencia: for i in $(seq 1 5); do ...
```

6.2. El Bucle `while`

Tema: Repetir tareas mientras una condición sea verdadera

Introducción teórica: El bucle `while` ejecuta un bloque de código repetidamente **mientras** la condición de prueba sea verdadera. Es ideal para leer archivos línea por línea, o para crear bucles infinitos (daemons) que se ejecutan hasta que se cumplen ciertas condiciones internas.

Ejemplo práctico (Lectura de archivo línea por línea):

```bash
#!/bin/bash
# Script para leer usuarios de un archivo

ARCHIVO_USUARIOS="usuarios.txt"

# Crear un archivo de ejemplo
echo "juan" > $ARCHIVO_USUARIOS
echo "pedro" >> $ARCHIVO_USUARIOS
echo "ana" >> $ARCHIVO_USUARIOS

# Leer el archivo línea por línea
while IFS= read -r LINEA; do
    echo "Procesando usuario: $LINEA"
done < "$ARCHIVO_USUARIOS"
```

Explicación línea por línea:

- `while IFS= read -r LINEA; do`: Es la forma estándar y robusta de leer un archivo línea por línea.
  - `IFS=`: Evita que los espacios iniciales/finales sean eliminados.
  - `read -r`: Lee una línea completa y la almacena en la variable `LINEA`. El flag `-r` evita que los caracteres de barra invertida sean interpretados.
- `done < "$ARCHIVO_USUARIOS"`: Redirige el contenido del archivo al bucle `while` como entrada estándar.

Ejercicio propuesto: Crea un bucle `while` que cuente de 10 hacia 0. El bucle debe detenerse cuando el contador sea menor que 0.

Resultado esperado:

```
Cuenta regresiva: 10
Cuenta regresiva: 9
...
Cuenta regresiva: 0
```

Solución (Fragmento de código):

```
CONTADOR=10
while [ $CONTADOR -ge 0 ]; do
    echo "Cuenta regresiva: $CONTADOR"
    CONTADOR=$((CONTADOR - 1)) # $((...)) es para operaciones aritméticas
done
```

Capítulo 7: Redirecciones, Tuberías y Filtros

7.1. Redirecciones de Entrada y Salida

Tema: Controlar el flujo de datos de los comandos

Introducción teórica: Cada comando en Bash tiene tres flujos de datos estándar:

1. **Entrada Estándar (stdin):** Descriptor de archivo `0`. De donde el comando lee datos (generalmente el teclado).
2. **Salida Estándar (stdout):** Descriptor de archivo `1`. Donde el comando escribe su resultado normal (generalmente la terminal).
3. **Error Estándar (stderr):** Descriptor de archivo `2`. Donde el comando escribe mensajes de error (generalmente la terminal).

Las **redirecciones** permiten cambiar el destino o la fuente de estos flujos.

Ejemplo práctico:

```
# 1. Redirigir la Salida Estándar a un archivo (sobrescribe)
ls -l /etc > listado.txt

# 2. Redirigir la Salida Estándar a un archivo (añade al final)
echo "Fin del listado" >> listado.txt

# 3. Redirigir el Error Estándar a un archivo
ls -l /ruta_inexistente 2> errores.log

# 4. Redirigir Salida y Error a un mismo archivo (método moderno)
ls -l /etc /ruta_inexistente &> salida_y_error.log

# 5. Redirigir la Entrada Estándar desde un archivo
wc -l < listado.txt
```

Explicación línea por línea:

- `> listado.txt`: Redirige `stdout` (1) al archivo, sobrescribiéndolo.
- `>> listado.txt`: Redirige `stdout` (1) al archivo, añadiendo al final.
- `2> errores.log`: Redirige `stderr` (2) al archivo.
- `&> salida_y_error.log`: Redirige ambos flujos (`stdout` y `stderr`) al archivo.
- `< listado.txt`: Redirige el contenido del archivo a `stdin` del comando `wc -l`.

Ejercicio propuesto: Ejecuta el comando `find /etc -name "*.conf"` (que lista archivos de configuración) y redirige la salida normal a un archivo llamado `conf_files.txt` y los errores (permisos denegados, etc.) a un archivo llamado `find_errors.txt`.

Resultado esperado: Dos archivos creados: `conf_files.txt` con el listado y `find_errors.txt` con los errores.

Solución:

```
find /etc -name "*.conf" > conf_files.txt 2> find_errors.txt
```

7.2. Tuberías (Pipes)

Tema: Conectar la salida de un comando con la entrada de otro

Introducción teórica: El operador de tubería (`|` , pipe) es uno de los conceptos más poderosos de Bash. Permite tomar la **Salida Estándar (`stdout`)** de un comando y usarla como la **Entrada Estándar (`stdin`)** del siguiente comando, creando una cadena de procesamiento de datos.

Ejemplo práctico:

```
# 1. Listar procesos, buscar los que contienen 'bash' y contar cuántos son
ps aux | grep bash | wc -l

# 2. Obtener la lista de archivos en /etc, ordenarla alfabéticamente y
mostrar solo las 10 primeras líneas
ls /etc | sort | head -n 10

# 3. Mostrar el contenido de un archivo y convertirlo a mayúsculas
cat listado.txt | tr '[:lower:]' '[:upper:]'
```

Explicación línea por línea:

- `ps aux` : Lista todos los procesos en ejecución.
- `| grep bash` : La salida de `ps aux` se convierte en la entrada de `grep` , que filtra las líneas que contienen la palabra “bash” .
- `| wc -l` : La salida de `grep` se convierte en la entrada de `wc -l` , que cuenta el número de líneas.
- `ls /etc | sort` : La lista de archivos se ordena alfabéticamente.
- `| head -n 10` : La lista ordenada se pasa a `head` , que muestra solo las 10 primeras líneas.
- `| tr '[:lower:]' '[:upper:]'` : El comando `tr` (translate) reemplaza todos los caracteres en minúscula por sus equivalentes en mayúscula.

Ejercicio propuesto: Muestra el contenido del archivo `/etc/passwd` , filtra las líneas que contienen la palabra “home” y luego cuenta cuántas de esas líneas existen.

Resultado esperado: Un número entero que representa la cantidad de usuarios con directorio “home”.

Solución:

```
cat /etc/passwd | grep home | wc -l
```

Capítulo 8: Procesamiento de Texto con `grep`, `sed` y `awk`

8.1. Filtrado con `grep`

Tema: Búsqueda de patrones en archivos

Introducción teórica: `grep` (Global Regular Expression Print) es la herramienta esencial para buscar líneas que coincidan con un patrón (generalmente una expresión regular) dentro de uno o más archivos.

Ejemplo práctico:

```
# 1. Buscar la palabra 'root' en el archivo /etc/passwd
grep root /etc/passwd

# 2. Buscar la palabra 'bash' ignorando mayúsculas/minúsculas (-i)
grep -i bash /etc/passwd

# 3. Mostrar las líneas que NO contienen la palabra 'nologin' (-v)
grep -v nologin /etc/passwd

# 4. Mostrar las líneas que coinciden y el número de línea (-n)
grep -n root /etc/passwd
```

Explicación de opciones:

- `-i`: Ignora mayúsculas y minúsculas.
- `-v`: Invierte la coincidencia (muestra las líneas que NO coinciden).

- `-n`: Muestra el número de línea donde se encontró la coincidencia.

Ejercicio propuesto: Busca en el archivo `/etc/passwd` todas las líneas que no contengan la palabra "false" y cuenta cuántas son.

Resultado esperado: Un número entero.

Solución:

```
grep -v false /etc/passwd | wc -l
```

8.2. Edición de Flujo con `sed`

Tema: Sustitución y edición de texto en el flujo de datos

Introducción teórica: `sed` (Stream Editor) es un editor de texto no interactivo que se utiliza principalmente para realizar transformaciones básicas de texto en el flujo de datos. Su uso más común es la sustitución de texto.

Ejemplo práctico (Sustitución):

```
# 1. Sustituir la primera ocurrencia de 'bin' por 'BIN' en cada línea
echo "bin/bash" | sed 's/bin/BIN/'

# 2. Sustituir TODAS las ocurrencias de 'user' por 'admin' en cada línea
(flag 'g')
echo "user user user" | sed 's/user/admin/g'

# 3. Eliminar líneas que contengan la palabra 'test'
cat /etc/passwd | grep -i test | sed '/test/d'
```

Explicación de la sintaxis `s/patron/reemplazo/flags`:

- `s`: Indica la operación de sustitución.
- `patron`: La expresión regular a buscar.
- `reemplazo`: El texto que sustituirá al patrón.
- `g`: Flag global, para sustituir todas las ocurrencias en la línea.

- `/patron/d`: Elimina (`d`) la línea que coincida con el patrón.

Ejercicio propuesto: Muestra el contenido del archivo `/etc/passwd` y sustituye todas las ocurrencias de `/bin/bash` por `/bin/zsh`.

Resultado esperado: El contenido de `/etc/passwd` con los shells de Bash cambiados a Zsh.

Solución:

```
cat /etc/passwd | sed 's/\/bin\/bash/\/bin\/zsh/g'
```

8.3. Procesamiento de Columnas con `awk`

Tema: Análisis y manipulación de datos estructurados por columnas

Introducción teórica: `awk` es un lenguaje de programación de propósito especial diseñado para el procesamiento de texto basado en patrones y acciones. Es excelente para trabajar con datos tabulares o archivos donde la información está separada por delimitadores (como el archivo `/etc/passwd`, que usa dos puntos `:`).

Ejemplo práctico:

```
# 1. Mostrar solo la primera columna (el nombre de usuario) del archivo
/etc/passwd
awk -F: '{ print $1 }' /etc/passwd

# 2. Mostrar el nombre de usuario ($1) y el shell ($7)
awk -F: '{ print "Usuario:", $1, "Shell:", $7 }' /etc/passwd

# 3. Mostrar solo las líneas donde el ID de usuario ($3) es menor a 1000
awk -F: '$3 < 1000 { print $1, $3 }' /etc/passwd
```

Explicación de la sintaxis:

- `awk -F:`: El flag `-F` define el delimitador de campo (Field Separator), en este caso, los dos puntos.

- `{ print $1 }`: La acción a realizar. `$1` es la primera columna, `$2` la segunda, y así sucesivamente.
- `$3 < 1000`: Es un patrón de condición. La acción `{ print $1, $3 }` solo se ejecuta si la tercera columna es menor a 1000.

Ejercicio propuesto: Usando `awk`, muestra el nombre de usuario (`$1`) y el directorio personal (`$6`) de todos los usuarios en `/etc/passwd` cuyo shell (`$7`) sea `/bin/bash`.

Resultado esperado:

```
root /root
usuario_ejemplo /home/usuario_ejemplo
...
```

Solución:

```
awk -F: '$7 == "/bin/bash" { print $1, $6 }' /etc/passwd
```

Resumen de Comandos y Conceptos Importantes (Nivel Intermedio)

Comando/Concepto	Descripción	Uso Principal	Ejemplo Clave
`if`	Estructura condicional.	Tomar decisiones.	`if [ $A -gt 10 ]; then ...`
`-gt`, **`-eq`**, **`-f`**	Operadores de comparación.	Evaluar números y archivos.	`[ $A -eq 5 ]`, `[ -f archivo.txt ]`
`case`	Estructura de selección múltiple.	Manejar opciones de menú.	`case $OPCION in a) ... esac`
`for`	Bucle de iteración.	Recorrer listas de archivos o números.	`for i in 1 2 3; do ...`
`while`	Bucle condicional.	Leer archivos línea por línea.	`while read LINEA; do ...`
`>`	Redirección de `stdout` (sobrescribe).	Guardar la salida de un comando.	`ls > lista.txt`
`>>`	Redirección de `stdout` (añade).	Añadir logs al final de un archivo.	`echo "log" >> log.txt`
`2>`	Redirección de `stderr`.	Capturar mensajes de error.	`comando 2> error.log`
`	`** | Tubería (Pipe). | Encadenar comandos. |` ps aux	grep bash`	
`grep`	Filtrado de texto.	Buscar patrones en archivos.	`grep -i palabra archivo`
`sed`	Edición de flujo.	Sustitución de texto.	`sed 's/viejo/nuevo/g'`

Comando/Concepto	Descripción	Uso Principal	Ejemplo Clave
`awk`	Procesamiento de columnas.	Análisis de datos tabulares.	`awk -F: '{ print $1 }'`

Ejercicio Integrador (Nivel Intermedio)

Objetivo: Crear un script de monitoreo de espacio en disco con alertas condicionales.

Instrucciones:

1. Crea un script llamado `monitor_disco.sh`.
2. El script debe usar el comando `df -h` para obtener el uso de disco.
3. Usa `grep` y `awk` para extraer el porcentaje de uso de la partición raíz (`/`).
4. Usa una estructura `if` para verificar si el porcentaje de uso es mayor al 80%.
5. Si es mayor, imprime una alerta: "ALERTA: Uso de disco en / es crítico: [Porcentaje]%".
6. Si es menor o igual, imprime: "Uso de disco en / es normal: [Porcentaje]%".

Solución (Script `monitor_disco.sh`):

```bash
#!/bin/bash
# Script de Monitoreo de Disco

# 1. Obtener el porcentaje de uso de la partición raíz (/)
# df -h: Muestra el uso de disco en formato legible
# grep '/$': Filtra la línea de la partición raíz
# awk '{print $5}' : Imprime la quinta columna (el porcentaje)
# tr -d '%': Elimina el signo de porcentaje para poder comparar como número

USO_DISCO=$(df -h | grep '/$' | awk '{print $5}' | tr -d '%')

# 2. Definir el umbral de alerta
UMBRAL=80

# 3. Estructura condicional para la alerta
if [ $USO_DISCO -gt $UMBRAL ]; then
    echo "ALERTA: Uso de disco en / es crítico: $USO_DISCO%"
else
    echo "Uso de disco en / es normal: $USO_DISCO%"
fi
```

Comandos de ejecución:

```bash
chmod +x monitor_disco.sh
./monitor_disco.sh
```

Nivel Avanzado: Funciones, Depuración y Automatización

Este nivel te llevará a dominar técnicas de programación más estructuradas, a escribir scripts robustos y a automatizar tareas críticas del sistema.

Capítulo 9: Funciones y Modularización de Scripts

9.1. Definición y Uso de Funciones

Tema: Reutilizar bloques de código

Introducción teórica: Las **funciones** permiten agrupar comandos en un bloque de código reutilizable. Esto mejora la legibilidad, reduce la repetición de código (principio DRY: Don’ t Repeat Yourself) y facilita la modularización de scripts complejos. Una función se define y luego se llama por su nombre.

Ejemplo práctico:

```bash
#!/bin/bash
# Script con una función para crear un respaldo

# Directorio de destino para el respaldo
DIR_RESPALDO="/tmp/respaldos_$(date +%Y%m%d)"

# Definición de la función
crear_respaldo() {
    local ORIGEN=$1 # Argumento 1: Directorio a respaldar
    local DESTINO=$2 # Argumento 2: Directorio de destino

    # 1. Verificar si el directorio de origen existe
    if [ ! -d "$ORIGEN" ]; then
        echo "Error: El directorio de origen '$ORIGEN' no existe."
        return 1 # Retorna un código de error
    fi

    # 2. Crear el directorio de destino si no existe
    mkdir -p "$DESTINO"

    # 3. Copiar el contenido
    cp -r "$ORIGEN" "$DESTINO"

    echo "Respaldo de '$ORIGEN' creado exitosamente en '$DESTINO'."
    return 0 # Retorna un código de éxito
}

# Llamada a la función con argumentos
crear_respaldo ~/Documentos "$DIR_RESPALDO"

# Verificar el código de salida de la función
if [ $? -eq 0 ]; then
    echo "Operación de respaldo completada."
else
    echo "Operación de respaldo fallida."
fi
```

Explicación línea por línea:

- `crear_respaldo() { ... }`: Sintaxis estándar para definir una función.
- `local ORIGEN=$1`: La palabra clave `local` es crucial. Asegura que la variable `ORIGEN` solo exista dentro de la función, evitando conflictos con variables globales del mismo nombre. `$1` es el primer argumento pasado a la función.

- `return 1`: Las funciones en Bash pueden devolver un código de salida (status) usando `return`. Al igual que los comandos, `0` es éxito y cualquier otro valor es error.
- `crear_respaldo ~/Documentos "$DIR_RESPALDO"`: Así se llama a la función, pasándole los argumentos.
- `if [ $? -eq 0 ]; then`: Se verifica el código de salida de la función (almacenado en la variable especial `$?`).

Ejercicio propuesto: Crea una función llamada `saludar` que reciba un nombre como argumento y lo imprima. Luego, llama a la función con tu nombre.

Resultado esperado:

```
¡Hola, [Tu Nombre]! Bienvenido a las funciones de Bash.
```

Solución (Fragmento de código):

```
saludar() {
    echo "¡Hola, $1! Bienvenido a las funciones de Bash."
}

saludar "Manus"
```

9.2. Paso de Argumentos y Variables Locales

Tema: Gestionar la información dentro y fuera de las funciones

Introducción teórica: El manejo de variables locales y el paso de argumentos son esenciales para escribir funciones robustas. Los argumentos se pasan a la función de la misma manera que se pasan a un script, y se acceden dentro de la función con `$1`, `$2`, `$#`, etc.

Ejemplo práctico (Uso de variables locales y globales):

```bash
#!/bin/bash
# Script que demuestra el alcance de las variables

VARIABLE_GLOBAL="Soy Global"

mostrar_variables() {
    local VARIABLE_LOCAL="Soy Local"
    echo "Dentro de la función:"
    echo "  - Global: $VARIABLE_GLOBAL"
    echo "  - Local: $VARIABLE_LOCAL"
}

mostrar_variables

echo "Fuera de la función:"
echo "  - Global: $VARIABLE_GLOBAL"
echo "  - Local: $VARIABLE_LOCAL" # Esta línea fallará, la variable local no existe aquí
```

Explicación:

- La `VARIABLE_GLOBAL` es accesible tanto dentro como fuera de la función.
- La `VARIABLE_LOCAL` solo existe dentro del bloque de la función `mostrar_variables`. Cuando se intenta acceder a ella fuera, su valor es nulo. **Siempre usa `local`** para variables internas de la función.

Ejercicio propuesto: Crea una función llamada `sumar` que reciba dos números como argumentos y use una variable local para almacenar el resultado de la suma. La función debe imprimir el resultado.

Resultado esperado (al llamar `sumar 15 20`):

```
La suma es: 35
```

Solución (Fragmento de código):

```bash
sumar() {
    local NUM1=$1
    local NUM2=$2
    local RESULTADO=$((NUM1 + NUM2))
    echo "La suma es: $RESULTADO"
}

sumar 15 20
```

Capítulo 10: Depuración y Buenas Prácticas de Scripting

10.1. Técnicas de Depuración (Debugging)

Tema: Encontrar y corregir errores en scripts

Introducción teórica: Los scripts complejos inevitablemente contendrán errores (bugs). Bash ofrece herramientas integradas para ayudarte a rastrear la ejecución del script y ver exactamente qué comandos se están ejecutando y con qué valores de variables.

Ejemplo práctico (Modos de depuración):

```bash
#!/bin/bash
# Script con un error intencional

echo "Inicio del script"

# Error: Intentar sumar una cadena de texto
RESULTADO=$((5 + "cinco"))

echo "Fin del script"
```

Comandos de depuración:

Comando	Descripción	Uso
`bash -n script.sh`	**No ejecutar.** Solo verifica la sintaxis del script.	Útil para errores de sintaxis.
`bash -v script.sh`	**Verbosidad.** Muestra las líneas de entrada del script a medida que se leen.	Útil para ver la secuencia de comandos.
`bash -x script.sh`	**Rastreo.** Muestra cada comando expandido (con variables sustituidas) antes de ejecutarlo, precedido por `+`.	El más útil para ver el flujo de ejecución y los valores de las variables.

Ejecución con rastreo (`-x`):

```
$ bash -x script_con_error.sh
+ echo 'Inicio del script'
Inicio del script
+ RESULTADO=5 + cinco
script_con_error.sh: línea 6: cinco: valor demasiado grande para la base
(error simbólico)
+ echo 'Fin del script'
Fin del script
```

Análisis del rastreo: La línea `+ RESULTADO=5 + cinco` muestra que Bash intentó realizar la operación aritmética, revelando que la variable `"cinco"` no es un número, lo que causa el error.

Ejercicio propuesto: Ejecuta el script de monitoreo de disco (`monitor_disco.sh` del Capítulo 8) usando el modo de rastreo (`-x`) y observa cómo se expanden las variables `USO_DISCO` y `UMBRAL` antes de la comparación `if`.

Resultado esperado (parte del rastreo):

```
+ UMBRAL=80
+ '[' 10 -gt 80 ']' # El 10 sería el valor de USO_DISCO
+ echo 'Uso de disco en / es normal: 10%'
```

Solución:

```
bash -x monitor_disco.sh
```

10.2. Buenas Prácticas de Scripting

Tema: Escribir código limpio, seguro y robusto

Introducción teórica: Un script profesional no solo funciona, sino que es fácil de leer, mantener y es resistente a errores inesperados. Adoptar buenas prácticas desde el inicio te ahorrará horas de depuración.

Buenas Prácticas Esenciales:

Práctica	Descripción	Ejemplo
Usar Shebang	Siempre incluir `#!/bin/bash` al inicio.	`#!/bin/bash`
Comentarios	Explicar la lógica del código con `#`.	`# Esta función valida la entrada`
Comillas Dobles	Siempre usar comillas dobles (`"`) alrededor de las variables para evitar problemas con espacios o caracteres especiales.	`mkdir "$DIR_NOMBRE"` (en lugar de `mkdir $DIR_NOMBRE`)
Variables Locales	Usar `local` dentro de las funciones.	`local CONTADOR=0`
Modo Estricto	Usar `set -euo pipefail` al inicio del script.	`set -euo pipefail`

El Modo Estricto (`set -euo pipefail`):

- `set -e` : Sale inmediatamente si un comando falla (código de salida distinto de cero).
- `set -u` : Trata las variables no definidas como un error y sale.
- `set -o pipefail` : Si un comando en una tubería (`|`) falla, el código de salida de toda la tubería será el del comando fallido (por defecto, solo se usa el código del último comando).

Ejemplo práctico (Script robusto):

```bash
#!/bin/bash
# Script robusto con modo estricto

set -euo pipefail

# 1. Función que falla si no recibe un argumento
validar_argumento() {
    if [ $# -ne 1 ]; then
        echo "Error: Se requiere un argumento." >&2
        exit 1
    fi
}

# 2. Uso de comillas dobles
ARCHIVO="mi archivo con espacios.txt"
touch "$ARCHIVO"

# 3. Uso de variable no definida (fallará por 'set -u')
# echo $VARIABLE_NO_DEFINIDA

echo "Script finalizado con éxito."
```

Ejercicio propuesto: Modifica el script `saludo.sh` del Capítulo 3 para que use `set -u`. Luego, intenta ejecutarlo sin pasarle ningún argumento. Observa cómo el script falla inmediatamente debido a que `$1` no está definido.

Resultado esperado (al ejecutar sin argumentos):

```
./saludo.sh: línea 5: $1: parámetro no establecido
```

Solución (Script `saludo.sh` modificado):

```bash
#!/bin/bash
set -u # Activa el modo estricto para variables no definidas
echo "Hola $1, bienvenido al curso de Bash."
```

Capítulo 11: Automatización de Tareas con `cron` y `at`

11.1. Tareas Recurrentes con `cron`

Tema: Programar scripts para que se ejecuten automáticamente

Introducción teórica: Cron es el demonio de programación de tareas más utilizado en sistemas Unix/Linux. Permite ejecutar comandos o scripts de forma recurrente en intervalos de tiempo específicos (minutos, horas, días, meses). Las tareas se definen en un archivo llamado **crontab**.

Estructura de una línea de Crontab: Una línea de crontab tiene 6 campos:

Campo	Rango de Valores	Descripción
1	0-59	Minuto
2	0-23	Hora
3	1-31	Día del mes
4	1-12	Mes
5	0-7	Día de la semana (0 o 7 es Domingo)
6	Comando	El script o comando a ejecutar

Ejemplo práctico (Configuración de Crontab): Para editar tu crontab personal, usa el comando `crontab -e`.

```
 # Ejecutar el script de limpieza todos los días a las 3:30 AM
30 3 * * * /home/usuario/scripts/limpieza.sh

# Ejecutar el script de monitoreo cada 15 minutos
*/15 * * * * /home/usuario/scripts/monitor_disco.sh

# Ejecutar un comando solo los lunes a las 9:00 AM
0 9 * * 1 echo "Es lunes, hora de trabajar" >> ~/lunes.log
```

Explicación de la sintaxis:

- `30 3 * * *`: Se ejecuta a los 30 minutos de la hora 3 (3:30 AM), cualquier día del mes, cualquier mes, cualquier día de la semana.
- `*/15 * * * *`: El `*/15` en el campo de minutos significa "cada 15 minutos".
- `0 9 * * 1`: Se ejecuta a los 0 minutos de la hora 9 (9:00 AM), cualquier día del mes, cualquier mes, pero solo el día 1 (Lunes).

Ejercicio propuesto: Escribe la línea de crontab necesaria para ejecutar un script de respaldo (`/home/usuario/backup.sh`) el primer día de cada mes a las 23:00 (11 PM).

Resultado esperado:

```
0 23 1 * * /home/usuario/backup.sh
```

Solución:

```
# 0 minutos, 23 horas, 1er día del mes, cualquier mes, cualquier día de la semana
0 23 1 * * /home/usuario/backup.sh
```

11.2. Tareas Únicas con `at`

Tema: Programar un comando para una sola ejecución futura

Introducción teórica: El comando `at` se utiliza para programar comandos o scripts para que se ejecuten **una sola vez** en un momento específico en el futuro. Es ideal para tareas que no necesitan repetirse.

Ejemplo práctico:

```
# Programar un mensaje para dentro de 5 minutos
echo "echo '¡Hora del café!' > ~/recordatorio.txt" | at now + 5 minutes

# Programar un reinicio del servidor para mañana a las 2:00 AM
echo "sudo reboot" | at 2:00 AM tomorrow
```

Explicación:

- `at now + 5 minutes`: Programa la tarea para dentro de 5 minutos.
- `at 2:00 AM tomorrow`: Programa la tarea para las 2:00 AM del día siguiente.
- `atq`: Muestra la cola de tareas pendientes.
- `atrm [número_de_trabajo]`: Elimina una tarea de la cola.

Ejercicio propuesto: Programa un comando para que se ejecute a las 18:30 de hoy (6:30 PM) y que cree un archivo llamado `fin_jornada.txt` en tu directorio personal.

Resultado esperado (comando a ejecutar):

```
echo "touch ~/fin_jornada.txt" | at 18:30
```

Capítulo 12: Manejo de Señales y Procesos en Segundo Plano

12.1. Procesos en Segundo Plano y Primer Plano

Tema: Ejecutar tareas sin bloquear la terminal

Introducción teórica: Un **proceso en primer plano** (`foreground`) toma el control de la terminal. Un **proceso en segundo plano** (`background`) se ejecuta sin bloquear la terminal, permitiéndote seguir trabajando.

Ejemplo práctico:

```
# 1. Ejecutar un script que tarda mucho en segundo plano
./script_largo.sh &

# 2. Ver la lista de trabajos en segundo plano
jobs

# 3. Detener un proceso en primer plano (lo envía a segundo plano detenido)
# Presiona Ctrl+Z mientras el proceso se ejecuta

# 4. Reanudar el último proceso detenido en segundo plano
bg

# 5. Traer el último proceso de segundo plano a primer plano
fg
```

Explicación:

- `&`: El ampersand al final del comando lo ejecuta inmediatamente en segundo plano.
- `jobs`: Muestra los trabajos que están en segundo plano o detenidos en la sesión actual.
- `Ctrl+Z`: Envía la señal `SIGTSTP` al proceso, deteniéndolo.
- `bg`: Reanuda el último trabajo detenido en segundo plano.
- `fg`: Trae el último trabajo de segundo plano a primer plano.

12.2. Captura de Señales con `trap`

Tema: Responder a eventos del sistema (como Ctrl+C)

Introducción teórica: Las **señales** son notificaciones asíncronas enviadas a un proceso para informarle de un evento. Por ejemplo, `Ctrl+C` envía la señal `SIGINT` (Interrupción). El comando `trap` permite a un script "capturar" estas señales y ejecutar un comando o función en lugar de la acción por defecto (que a menudo es terminar).

Ejemplo práctico (Captura de SIGINT):

```bash
#!/bin/bash
# Script que captura Ctrl+C

# Función que se ejecuta al recibir la señal SIGINT
limpiar_y_salir() {
    echo -e "\n[INFO] Señal SIGINT (Ctrl+C) capturada."
    echo "[INFO] Realizando limpieza de archivos temporales..."
    rm -f /tmp/temp_*.log
    echo "[INFO] Limpieza completa. Saliendo."
    exit 1 # Salir con código de error
}

# 1. Configurar el trap: Ejecutar la función 'limpiar_y_salir' al recibir SIGINT (2)
trap limpiar_y_salir INT

# 2. Crear un archivo temporal para la demostración
touch /tmp/temp_1.log

echo "El script se está ejecutando. Presiona Ctrl+C para detenerlo de forma segura."
# Bucle infinito para mantener el script vivo
while true; do
    sleep 1
done
```

Explicación:

- `trap limpiar_y_salir INT`: Le dice a Bash que cuando reciba la señal `INT` (que corresponde al número 2), ejecute la función `limpiar_y_salir`.
- `exit 1`: Es importante salir del script dentro de la función `trap`, ya que de lo contrario el script continuaría ejecutándose.

Ejercicio propuesto: Modifica el script anterior para que, además de `SIGINT` (2), también capture la señal `SIGTERM` (15), que es la señal de terminación enviada por el comando `kill` por defecto.

Resultado esperado: La función `limpiar_y_salir` se ejecuta si se presiona `Ctrl+C` o si se ejecuta `kill [PID_del_script]`.

Solución (Fragmento de código):

```bash
# Capturar SIGINT (2) y SIGTERM (15)
trap limpiar_y_salir INT TERM
```

Resumen de Comandos y Conceptos Importantes (Nivel Avanzado)

Comando/Concepto	Descripción	Uso Principal	Ejemplo Clave
Función	Bloque de código reutilizable.	Modularización de scripts.	`mi_func() { ... }`
`local`	Define una variable con alcance local.	Evitar conflictos de variables.	`local VAR="valor"`
`return`	Devuelve un código de salida (status).	Indicar éxito (0) o fallo (>0).	`return 1`
`set -e`	Sale al primer error.	Robustez del script.	`set -e`
`set -u`	Error si se usa variable no definida.	Evitar errores silenciosos.	`set -u`
`set -o pipefail`	Falla si un comando en el pipe falla.	Depuración de tuberías.	`set -o pipefail`
`bash -x`	Modo de rastreo (debugging).	Ver la ejecución expandida.	`bash -x script.sh`
`cron`	Programador de tareas recurrentes.	Automatización diaria/mensual.	`0 1 * * * comando`
`at`	Programador de tareas únicas.	Ejecución puntual en el futuro.	`echo comando | at 10:00`
`&`	Ejecuta un proceso en segundo plano.	No bloquear la terminal.	`./script.sh &`
`trap`	Captura señales del sistema.	Manejo seguro de interrupciones.	`trap 'cleanup' INT`

Nivel Experto: Proyecto Integrador

El objetivo de este nivel es aplicar todos los conocimientos adquiridos (desde el manejo de archivos hasta funciones, condicionales y automatización) para construir una solución real y funcional.

Capítulo 13: Proyecto Integrador: Sistema de Respaldo Automatizado

13.1. Requisitos del Sistema

Tema: Diseño de un script de respaldo profesional

Introducción teórica: Un sistema de respaldo (backup) es crucial en la administración de sistemas. Nuestro script debe ser capaz de:

1. Recibir un directorio de origen como argumento.
2. Crear un directorio de destino con fecha.
3. Comprimir el directorio de origen en un archivo `.tar.gz`.
4. Verificar el éxito de la operación.
5. Limpiar los archivos temporales.
6. Ser ejecutable mediante `cron`.

13.2. Script de Respaldo (`backup_system.sh`)

Ejemplo práctico (Script completo):

```bash
#!/bin/bash
# Manual Completo de Bash: De Cero a Experto
# Proyecto Integrador: Sistema de Respaldo Automatizado

# --- Configuración de Robustez ---
set -euo pipefail

# --- Variables Globales ---
FECHA=$(date +%Y%m%d_%H%M%S)
DIR_BASE_RESPALDO="/var/backups/manual_bash"
LOG_FILE="$DIR_BASE_RESPALDO/backup.log"

# --- Funciones ---

# Función para registrar mensajes en el log
log_message() {
    local TIPO=$1
    local MENSAJE=$2
    echo "[$(date +%Y-%m-%d %H:%M:%S)] [$TIPO] $MENSAJE" | tee -a
"$LOG_FILE"
}

# Función de limpieza y manejo de errores (trap)
cleanup_and_exit() {
    local EXIT_CODE=$?
    if [ $EXIT_CODE -ne 0 ]; then
        log_message "ERROR" "El script terminó con código de error
$EXIT_CODE. Revisar log."
    else
        log_message "INFO" "Respaldo completado exitosamente."
    fi
    # Opcional: Eliminar archivos temporales si los hubiera
    # rm -f /tmp/temp_backup_*
    exit $EXIT_CODE
}

# Función principal de respaldo
perform_backup() {
    local ORIGEN=$1
    local NOMBRE_DIR=$(basename "$ORIGEN")
    local DIR_DESTINO="$DIR_BASE_RESPALDO/$NOMBRE_DIR"
    local ARCHIVO_RESPALDO="$DIR_DESTINO/${NOMBRE_DIR}_${FECHA}.tar.gz"

    log_message "INFO" "Iniciando respaldo de $ORIGEN..."
```

```
    # 1. Validar el directorio de origen
    if [ ! -d "$ORIGEN" ]; then
        log_message "FATAL" "Directorio de origen no encontrado: $ORIGEN"
        return 1
    fi

    # 2. Crear directorios de destino
    mkdir -p "$DIR_DESTINO"
    log_message "INFO" "Directorio de destino creado: $DIR_DESTINO"

    # 3. Crear el archivo comprimido (tar.gz)
    log_message "INFO" "Comprimiendo $ORIGEN a $ARCHIVO_RESPALDO..."
    tar -czf "$ARCHIVO_RESPALDO" -C "$(dirname "$ORIGEN")" "$(basename
"$ORIGEN")"

    # 4. Verificar el código de salida de tar
    if [ $? -eq 0 ]; then
        log_message "SUCCESS" "Archivo de respaldo creado: $(du -h
"$ARCHIVO_RESPALDO" | awk '{print $1}')"
    else
        log_message "ERROR" "Fallo al crear el archivo tar.gz."
        return 1
    fi
}

# --- Lógica Principal ---

# 1. Configurar el trap para que se ejecute al salir (EXIT)
trap cleanup_and_exit EXIT

# 2. Validar argumentos
if [ $# -ne 1 ]; then
    log_message "FATAL" "Uso: $0 <directorio_a_respaldar>"
    exit 1
fi

# 3. Ejecutar el respaldo
perform_backup "$1"

# El script terminará aquí y el trap EXIT se encargará de la salida final.
```

Explicación de la Solución:

- `set -euo pipefail`: Garantiza que el script se detenga ante cualquier error, haciendo el proceso más seguro.

- **`log_message`**: Una función modular que escribe mensajes tanto en la terminal (`tee -a`) como en el archivo de log, facilitando el seguimiento.
- **`trap cleanup_and_exit EXIT`**: Esta es la clave. La señal `EXIT` se dispara justo antes de que el script termine, independientemente de si fue por éxito o por un error (`set -e`). Esto asegura que la función `cleanup_and_exit` siempre se ejecute.
- **`tar -czf ...`**: El comando de compresión.
  - `-c`: Crear un nuevo archivo.
  - `-z`: Comprimir con gzip.
  - `-f`: Especificar el nombre del archivo.
  - `-C "$(dirname "$ORIGEN")"`: Cambia temporalmente al directorio padre del origen antes de comprimir, asegurando que el archivo comprimido solo contenga el directorio de origen y no toda la ruta absoluta.

13.3. Automatización Final

Tema: Poner el sistema de respaldo en producción

Instrucciones de Automatización:

1. Guarda el script como `backup_system.sh`.
2. Dale permisos de ejecución: `chmod +x backup_system.sh`.
3. Crea un directorio de prueba: `mkdir -p ~/datos_importantes`.
4. Edita tu crontab: `crontab -e`.
5. Añade la siguiente línea para ejecutar el respaldo todos los días a las 00:00 (medianoche):

```
# Ejecutar el respaldo de datos_importantes todos los días a medianoche
0 0 * * * /home/usuario/backup_system.sh /home/usuario/datos_importantes
```

Verificación: Después de la hora programada, verifica que el respaldo se haya creado y que el log contenga el mensaje de éxito:

```
ls -l /var/backups/manual_bash/datos_importantes/
cat /var/backups/manual_bash/backup.log
```

¡Felicidades! Has completado el **Manual Completo de Bash: De Cero a Experto**. Ahora tienes las herramientas y el conocimiento para automatizar tareas, administrar sistemas y escribir scripts profesionales en cualquier entorno Unix/Linux.

Anexo A: Resumen Consolidado de Comandos Esenciales

Esta tabla resume los comandos más importantes aprendidos a lo largo del manual, organizados por su función principal.

Categoría	Comando	Descripción	Nivel
Navegación	`pwd`	Muestra el directorio de trabajo actual.	Básico
	`ls`	Lista el contenido del directorio.	Básico
	`cd`	Cambia el directorio de trabajo.	Básico
Manipulación	`mkdir`	Crea un nuevo directorio.	Básico
	`touch`	Crea un archivo vacío.	Básico
	`cp`	Copia archivos y directorios.	Básico
	`mv`	Mueve o renombra archivos/directorios.	Básico
	`rm`	Elimina archivos (`-r` para directorios).	Básico
Entrada/Salida	`echo`	Imprime texto o variables.	Básico
	`read`	Captura la entrada del usuario.	Intermedio
	`>` / `>>`	Redirección de salida (sobrescribe / añade).	Intermedio
	`2>`	Redirección de error estándar.	Intermedio
	`|`	Tubería (Pipe) para encadenar comandos.	Intermedio
Control de Flujo	`if` / `elif` / `else`	Estructura condicional.	Intermedio
	`case`	Estructura de selección múltiple.	Intermedio
	`for`	Bucle de iteración sobre listas.	Intermedio
	`while`	Bucle condicional.	Intermedio
Procesamiento	`grep`	Busca patrones en texto.	Intermedio
	`sed`	Editor de flujo (sustitución).	Intermedio

Categoría	Comando	Descripción	Nivel
	`awk`	Procesamiento de columnas y datos tabulares.	Intermedio
	`tar`	Compresión y descompresión de archivos.	Avanzado
Scripting	`chmod +x`	Otorga permiso de ejecución a un script.	Básico
	`function` / `()`	Define una función.	Avanzado
	`local`	Define una variable de alcance local.	Avanzado
	`return`	Devuelve un código de salida.	Avanzado
	`set -euo pipefail`	Modo estricto para scripts robustos.	Avanzado
	`trap`	Captura señales del sistema.	Avanzado
Automatización	`crontab -e`	Edita las tareas programadas recurrentes.	Avanzado
	`at`	Programa una tarea para una sola ejecución futura.	Avanzado
	`&`	Ejecuta un proceso en segundo plano.	Avanzado

Anexo B: Espacio para Anotaciones y Comandos Personalizados

Utiliza este espacio para registrar tus propios comandos, trucos de terminal, o notas importantes que descubras durante tu práctica.

Mis Comandos Favoritos

```
# Ejemplo: Alias para actualizar el sistema
alias update='sudo apt update && sudo apt upgrade -y'

# Ejemplo: Función para crear un proyecto rápidamente
new_project() {
    mkdir "$1" && cd "$1" && git init
}
```

Notas Importantes

1. …
2. …
3. …
4. …
5. …