Shell Programming with the Bourne Again Shell (Bash)

Unit-IV

SCS281: Linux and Shell Programming

Mapped Course Outcomes (CO): CO3

What is Shell Scripting?

+ Shell scripting is writing a series of commands for the shell to execute.
+ It automates tasks that would otherwise be performed manually.
+ Used in Linux/Unix environments for system management, task automation, etc.

Advantages of Shell Scripting

+- Automates repetitive tasks

+- Simplifies complex system management

+- Enhances productivity and efficiency

+- Useful for system monitoring and backups

+- Portable across Unix/Linux environments

Common Shells

+- Bourne Shell (sh)
+- Bash (Bourne Again Shell)
+- Korn Shell (ksh)
+- C Shell (csh)
+- Z Shell (zsh)

Bourne Again Shell (Bash)-Introduction

- The Bourne Again Shell (Bash) is a Unix shell and command language, commonly used in Linux and macOS systems. Developed as a free and improved alternative to the original Bourne Shell (sh), Bash incorporates powerful scripting capabilities, command-line interaction, and support for advanced programming constructs.
- Bash is a cornerstone of many system administration tasks, allowing users to automate repetitive tasks, manage processes, and interact with the operating system efficiently.

Shell Responsibilities

A shell acts as an intermediary between the user and the operating system. It provides an interface for command execution, script processing, and system management. Key responsibilities of the shell include:

1. **Command Execution**
   - Accepts commands from the user or script and translates them into system calls.
   - Handles external programs and utilities as well as built-in commands (e.g., cd, echo, exit).
2. **Scripting and Automation**
   - Enables automation by supporting programming constructs like loops, conditionals, and functions.
   - Allows users to write scripts for tasks like backups, monitoring, and file management.

Shell Responsibilities

3. **Input/Output Redirection**
   - Redirects input (<) and output (>) streams to files or other commands.
   - Supports pipes (|) for chaining commands.
4. **Job and Process Management**
   - Manages processes, allowing users to start, stop, and monitor jobs.
   - Includes support for background (&) and foreground process control.
5. **Environment Management**
   - Handles environment variables (e.g., PATH, HOME) that influence program behavior.
   - Supports shell initialization scripts like .bashrc or .bash_profile for customization.

Shell Responsibilities

6. **Error Handling**
   - Provides mechanisms for error detection and recovery using exit status codes and conditional statements.
7. **Globbing and Pattern Matching**
   - Expands wildcard patterns (*, ?) in filenames for matching multiple files.
8. **User Interaction**
   - Offers features like command history, aliases, and auto-completion for an efficient user experience.

Features of Bash

- **Compatibility**: Backward compatible with the Bourne Shell (sh).
- **Programming Constructs**: Support for arrays, arithmetic operations, and string manipulation.
- **Customizability**: Users can define aliases, functions, and shell options to tailor their environment.
- **Portability**: Scripts written in Bash can run on various Unix-like systems with minimal modifications.

Pipes and Redirection in Bash

Pipes and redirection are fundamental concepts in Bash that allow you to control how data flows between commands, files, and the terminal.

These tools make Bash an incredibly powerful environment for handling data and automating tasks.

1. Pipes (|)

A pipe connects the output of one command to the input of another, allowing multiple commands to work together in a chain.

Syntax:

command1 | command2

Examples:

1. Using a Pipe to Count Lines in a File:

cat file.txt | wc -l

cat file.txt: Displays the content of file.txt.

wc -l: Counts the number of lines in the input received from cat.

2. Filtering Output:

ls -l | grep ".txt"

ls -l: Lists all files in the current directory.

grep ".txt": Filters the output to show only .txt files.

3. Combining Multiple Commands:

ps aux | grep "bash" | wc -l

Counts the number of processes related to bash.

2. Redirection

Redirection allows you to control the input and output of commands by connecting them to files or other streams.

Types of Redirection:

Standard Output (>): Redirects the output of a command to a file, overwriting the file if it exists.

echo "Hello, World!" > output.txt

Writes "Hello, World!" to output.txt.

Append Output (>>): Appends the output of a command to a file without overwriting.

echo "New line" >> output.txt

2. Redirection

Standard Input (<): Redirects input from a file to a command.

wc -l < file.txt

Counts the lines in file.txt.

Standard Error (2>): Redirects error messages to a file.

ls non_existing_file 2> error.log

Redirect Both Output and Error (>&): Redirects both standard output and standard error to the same destination.

command > output.log 2>&1

Discard Output (/dev/null): Sends the output or error to /dev/null to ignore it.

command > /dev/null 2>&1

Combining Pipes and Redirection

You can combine pipes and redirection to create more complex workflows.

Example:

cat file.txt | grep "keyword" > results.txt

- cat file.txt: Reads the file.
- grep "keyword": Filters lines containing "keyword".
- > results.txt: Saves the filtered lines into results.txt.

Practical Examples

Count Unique Words in a File:

```
cat file.txt | tr ' ' '\n' | sort | uniq -c | sort -nr > word_count.txt
```

Breaks the file into words, sorts them, counts unique occurrences, and saves the result.

Find Large Files:

```
du -ah | sort -rh | head -n 10 > large_files.txt
```

Lists the 10 largest files/directories in the current directory.

Quick Summary

Symbol	Meaning
>	Redirects standard output to a file.
>>	Appends standard output to a file.
<	Redirects standard input from a file.
2>	Redirects standard error to a file.
>&	Combines standard output and error streams.
`	`

Here Documents in Bash

- A *Here Document* is a way to pass multiline input to a command or script directly within a Bash script or terminal session.
- It is especially useful when you need to provide a block of text as input or when writing structured data, such as configuration files or SQL commands.

Here Documents in Bash

Syntax

command <<DELIMITER

text or commands

DELIMITER

- **command**: The command that processes the input.
- **DELIMITER**: A unique token indicating the start and end of the text block. Commonly used delimiters are EOF, END, or custom labels.
- **Text or Commands**: The block of text provided to the command.

The DELIMITER must appear on a new line and at the start of the line (no leading spaces) to signal the end of the input block.

Basic Example

```
cat <<EOF
Hello, World!
This is a here document example.
EOF
```

Output:

```
Hello, World! This is a here document example.
```

Use Cases

1. Writing Multiline Text to a File

You can redirect the output of a Here Document to a file using > or >>.

```
cat <<EOF > example.txt
Line 1
Line 2
Line 3
EOF
```

This creates (or overwrites) a file example.txt with the specified lines.

2. Passing Multiline Input to a Command

Here Documents can pass input to commands like mail, ftp, or sql.

```
mail -s "Test Email" user@example.com <<EOF
Dear User, This is a test email sent using a Here Document.
Regards,
Admin
EOF
```

3. Writing Inline Scripts

Use Here Documents to embed scripts or commands inline.

```
bash <<EOF
echo "Starting the script..."
mkdir test_dir
echo "Directory 'test_dir' created."
EOF
```

4. Writing Configuration Files

Here Documents are helpful for generating configuration files directly from a script.

```
cat <<EOF > config.conf
[General]
name=Example
version=1.0
[Settings]
enabled=true
EOF
```

Quoted vs Unquoted Delimiters

Unquoted Delimiters:

Variable and command substitution is **enabled**.

```
name="Alice"
cat <<EOF
Hello, $name!
Today is $(date).
EOF
```

Output:

```
Hello, Alice! Today is Mon Nov 18 12:00:00 UTC 2024.
```

Quoted Delimiters ('EOF' or "EOF"):

Variable and command substitution is **disabled**.

```
name="Alice"
cat <<'EOF’
Hello, $name!
Today is $(date).
EOF
```

Output:

```
Hello, $name! Today is $(date).
```

Ignoring Leading Tabs with <<-

If the Here Document text is indented, use <<-DELIMITER to strip leading tabs while preserving the structure.

```
cat <<-EOF
		This text
			is indented
			with tabs.
EOF
```

Error Handling

You can use Here Documents with commands that handle errors:

```
command <<EOF || echo "Command failed!"
input text
EOF
```

Summary Table

Feature	Example
Multiline text	cat <<EOF ... EOF
Save to file	cat <<EOF > file ... EOF
Variable substitution	cat <<EOF ... $var ... EOF
Disable substitution	cat <<'EOF' ... EOF
Strip leading tabs	cat <<-EOF ... EOF

Running a Shell Script

A **shell script** is a text file containing a sequence of commands, and it is executed by the shell.

Shell scripts automate repetitive tasks, perform system administration, and act as powerful tools for complex programming tasks.

Steps to Run a Shell Script

1. **Create the Script File**:
    1. Use a text editor to write a script.
    2. Example script (example.sh):

```
#!/bin/bash
echo "Hello, World!”
```

2. **Make the Script Executable**:

- Use the chmod command to give execution permissions:

```
chmod +x example.sh
```

Steps to Run a Shell Script

3. **Execute the Script**:

   Run the script using one of the following methods:

   - **Direct Execution**:

     ./example.sh

   - **Using the Shell**:

     bash example.sh

The Shell as a Programming Language

The shell is a powerful programming language that allows developers to write scripts using programming constructs like variables, loops, conditionals, and functions.

Key Features of the Shell as a Programming Language

1. Variables:

Define and use variables:

```
name="Alice"
echo "Hello, $name"
```

2. Conditionals:

Perform decision-making using if, elif, and else:

```
if [ "$name" == "Alice" ]; then
    echo "Welcome, Alice!"
else
    echo "You are not Alice."
fi
```

Key Features of the Shell as a Programming Language

3. Loops:

Execute repetitive tasks:

```
for i in {1..5}; do
    echo "Iteration $i"
done
```

4. Functions:

Group commands for reuse:

```
greet() {
    echo "Hello, $1!"
}
greet "Alice"
```

Key Features of the Shell as a Programming Language

5. Error Handling:

Use exit codes and conditions:

```
if ! command; then
    echo "Command failed!"
fi
```

6. Input/Output and File Handling:

Read/write files and manage input/output streams:

```
echo "Data" > file.txt
cat file.txt
```

Shell Meta-Characters

Meta-characters are special symbols in the shell that have predefined meanings and functions. They help in command processing, input/output redirection, and pattern matching.

Common Shell Meta-Characters

Meta-Character	Description	Example
#	Starts a comment	# This is a comment
$	References a variable	echo $HOME
\	Escapes the following character	echo \$HOME
*	Matches zero or more characters (wildcard)	ls *.txt
?	Matches a single character	ls file?.txt
[]	Matches a range of characters	ls file[1-3].txt
{}	Brace expansion	echo {1..5}

Shell Meta-Characters

Meta-Character	Description	Example
`	`	Pipe, passes output of one command to another
>	Redirects standard output to a file (overwrite)	echo "data" > file.txt
>>	Appends output to a file	echo "data" >> file.txt
<	Redirects input from a file	wc -l < file.txt
&	Runs a command in the background	command &
;	Separates commands on the same line	echo A; echo B
&&	Executes the next command if the previous succeeded	command1 && command2

Shell Meta-Characters

Meta-Character	Description	Example
()	Groups commands in a subshell	(cd /tmp && ls)
[]	Conditional test expression	[-f file.txt]
`	Command substitution	echo "Today is: $(date)"
"	Weak quoting, allows variable and command expansion	echo "Hello, $USER"
'	Strong quoting, disables all expansions	echo 'Hello, $USER'

Examples of Meta-Characters in Use

Wildcard Expansion:

ls *.txt

Lists all .txt files.

Command Chaining:

mkdir test && cd test || echo "Failed to create directory"

Creates a directory and navigates to it if successful, otherwise displays an error message.

Redirection:

echo "Data" > file.txt

cat file.txt

Background Execution:

sleep 10 &

File Name Substitution in Bash

File name substitution (also known as **globbing**) allows you to use wildcards to match file and directory names. It simplifies operations on multiple files and directories.

Wildcards and Patterns

Symbol	Description	Example
*	Matches zero or more characters	ls *.txt (all .txt files)
?	Matches exactly one character	ls file?.txt (e.g., file1.txt)
[abc]	Matches any one character from the set	ls file[abc].txt
[a-z]	Matches a range of characters	ls file[a-z].txt
[^abc]	Matches characters not in the set	ls file[^a-c].txt
{}	Matches comma-separated patterns	ls {file1,file2}.txt

Examples:

Match All Files with a Certain Extension:

ls *.txt

Match Files Starting with "log" and Ending with Any Character:

ls log?.txt

Match Files with Specific Patterns:

ls file[1-3].txt

Shell Variables

Shell variables store data for use in scripts or commands. These variables can hold strings, numbers, or command outputs.

Define a Variable:

my_var="Hello, World!"

Access a Variable:

echo $my_var

Unset a Variable:

unset my_var

Read Input into a Variable:

read user_name

echo "Welcome, $user_name!"

Types of Variables

Local Variables: Available only within the current shell or script.

local_var="local"

Environment Variables: Accessible by the shell and its child processes.

export PATH="/usr/local/bin:$PATH"

Examples:

Using a Variable in a Script:

```
name="Alice"
echo "Hello, $name!"
```

Environment Variable Example:

```
export MY_VAR="Persistent Data"
echo $MY_VAR
```

Command Substitution

Command substitution allows the output of a command to be assigned to a variable or used as an argument in another command.

Syntax:

Using backticks:

```
result=`command`
```

Using $() (preferred):

```
result=$(command)
```

Examples:

Assign Command Output to a Variable:

```
current_date=$(date)
echo "Today's date is: $current_date"
```

Use Command Output Inline:

```
echo "There are $(ls | wc -l) files in the directory."
```

Capture Output of a Command:

```
hostname=$(hostname)
echo "This machine's hostname is $hostname"
```

Nested Command Substitution:

Command substitution can be nested to process data further:

```
echo "The first file is: $(ls | head -1)"
```

Comparison Table

Feature	Description	Example
File Name Substitution	Matches files using patterns or wildcards	ls *.txt
Shell Variables	Stores and retrieves data in the shell	name="Alice"; echo $name
Command Substitution	Embeds command output into another command	date=$(date); echo $date

Real-World Use Case

```
#!/bin/bash
# Count .txt files and display the largest one

txt_count=$(ls *.txt 2>/dev/null | wc -l)
largest_file=$(ls -S *.txt 2>/dev/null | head -1)

if [ $txt_count -gt 0 ]; then
    echo "There are $txt_count .txt files."
    echo "The largest file is: $largest_file"
else
    echo "No .txt files found."
fi
```

Shell Commands

Shell commands are instructions that interact with the operating system to perform various tasks. They include built-in shell commands, external utilities, and custom scripts.

Types of Shell Commands

Built-in Commands: Integrated into the shell.

- Example: cd, echo, exit, read.
- Usage: cd /home changes the current directory.

External Commands: Executable programs in the system.

- Example: ls, grep, awk, sed.
- Usage: ls -l lists files in the current directory in long format.

Shell Commands

Shell Scripts: Custom scripts that automate tasks.

- Example:

```
#!/bin/bash
echo "Hello, World!"
```

Aliases: Shortcuts for frequently used commands.

Example:

```
alias ll='ls -l'
```

Examples:

Basic Command:

```
echo "Hello, Bash!"
```

Pipeline:

```
ls | grep ".txt"
```

The Environment

The environment refers to the settings and variables the shell uses to configure its behavior and interaction with the system.

Environment Variables

Environment variables are key-value pairs that influence the behavior of processes.

Environment Variables

Variable	Description
PATH	Directories to search for commands.
HOME	The user's home directory.
USER	The current user's name.
PWD	Current working directory.
SHELL	The default shell.

Commands for Managing Environment Variables

View All Variables:

printenv

Display a Specific Variable:

echo $HOME

Set a Variable Temporarily:

export MY_VAR="Hello"

echo $MY_VAR

Unset a Variable:

unset MY_VAR

Quoting

Quoting is used to control how special characters are interpreted by the shell.

Types of Quotes

Single Quotes (')

Preserves the literal value of all characters.

Example:

```
echo 'Hello $USER’
# Output: Hello $USER
```

Quoting

Double Quotes ("):

Allows variable and command substitution.

Example:

```
echo "Hello $USER"
# Output: Hello <username>
```

Backslash (\):

Escapes the next character.

Example:

```
echo Hello\ World
# Output: Hello World
```

Backticks (`) or $(...):

Command substitution.

Example:

```
echo "Today is $(date)"
```

The test Command

The test command evaluates conditional expressions and returns an exit status:

0 for true.

1 for false.

Syntax

test expression

or

[expression]

Common Test Expressions

Type	Expression	Example
File Tests		
File exists	-e file	[-e myfile.txt]
Is a regular file	-f file	[-f myfile.txt]
Is a directory	-d dir	[-d mydir]
Is readable	-r file	[-r myfile.txt]
String Tests		
Strings are equal	string1 = string2	["$USER" = "root"]
String is non-empty	-n string	[-n "$USER"]
String is empty	-z string	[-z "$password"]
Numeric Tests		
Numbers are equal	n1 -eq n2	[5 -eq 5]
Greater than	n1 -gt n2	[10 -gt 5]
Less than	n1 -lt n2	[3 -lt 5]

Examples

Basic Test Usage:

```
if [ -f myfile.txt ]; then
    echo "File exists!"
else
    echo "File does not exist."
fi
```

String Test:

```
name="Alice"
if [ "$name" = "Alice" ]; then
    echo "Hello, Alice!"
else
    echo "Who are you?"
fi
```

Numeric Test:

```
x=10
if [ $x -gt 5 ]; then
    echo "$x is greater than 5"
fi
```

Logical Operators:

AND (&&):

```
if [ -f file1.txt ] && [ -f file2.txt ]; then
    echo "Both files exist."
fi
```

OR (||):

```
if [ -f file1.txt ] || [ -f file2.txt ]; then
    echo "At least one file exists."
fi
```

Comparison Table

Feature	Example	Description
Shell Commands	ls, cd, grep	Basic utilities for system tasks.
Environment Variables	echo $HOME	Configure shell behavior.
Quoting	'text', "text", \$var	Control interpretation of special characters.
Test Command	[-f file], [$x -gt 5]	Evaluate expressions for conditions.