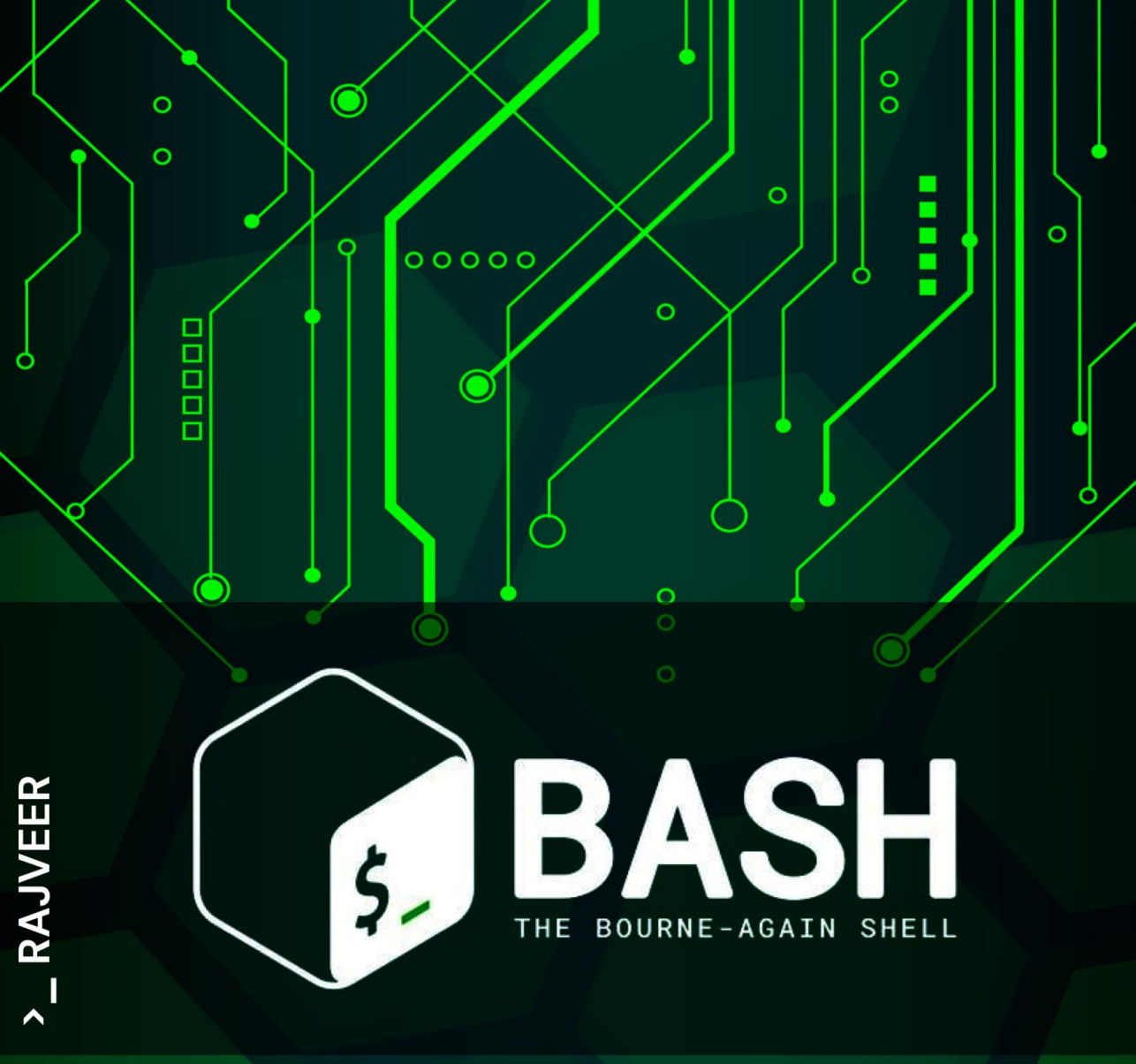

>_ RAJVEER

BASH

THE BOURNE-AGAIN SHELL

#!/bash-basics

For Linux lovers

Bash Basics

Intro to Bash

Bash (Bourne Again Shell) is a Unix shell and command language, widely used as the default login shell for most Linux distributions and macOS. It's an improved version of the original Bourne Shell (sh) and includes a number of features inspired by other shells, such as the Korn Shell (ksh) and the C Shell (csh).

1. What is a Shell?

A shell is a command-line interpreter that provides a user interface for accessing the services of an operating system. Shells allow users to execute commands, run scripts, and manage system processes. Bash is one of many available shells, but it's one of the most commonly used due to its balance of functionality and ease of use.

2. Bash vs. Other Shells

- **Bash**: Known for its simplicity, scripting capability, and widespread use.

- **Zsh**: Similar to Bash but with more advanced features like better autocompletion, theming, and plugins.
- **Ksh (Korn Shell)**: Offers advanced scripting features and is faster in execution for complex tasks.
- **Csh/Tcsh**: C-like syntax and advanced scripting capabilities, but less commonly used for everyday tasks.

“Hello world” in Bash

Creating a bash script is a straightforward process, but understanding how it works, especially the role of the shebang (`#!/bin/bash`), requires a bit of insight into how the Unix-like operating systems interpret and execute these scripts.

1. Creating a Bash Script

A bash script is simply a text file containing a series of commands that you could also type in the command line, but instead, they're stored in a file to be executed as a script.

Steps to Create a Bash Script:

1. **Create the Script File:**
   - You can use any text editor like `nano`, `vim`, or `gedit`.

   ```
   nano script.sh
   ```

   This command opens the nano editor, allowing you to create and edit the file named `script.sh`.

2. **Write the Script:**
   - Start by writing the commands you want to execute. It’s common practice to begin with a shebang line (`#!/bin/bash`).

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

- Save the file and exit the editor.

3. **Make the Script Executable:**
   - Before running the script, you need to make it executable.

```
chmod +x script.sh
```

4. **Run the Script:**
   - Execute the script from the terminal.

```
./script.sh
```

   This command will run the script and print `Hello, World!` to the terminal.

2. Understanding Shebang (`#!/bin/bash`)

The **shebang** (`#!`) is a character sequence at the very beginning of a script that tells the operating system which interpreter to use to execute the script.

- **Syntax:**

```
#!/path/to/interpreter
```

  In the case of bash scripts, the shebang typically looks like:

```
#!/bin/bash
```

  This line tells the system to use the Bash shell located at `/bin/bash` to interpret the script.

- **Without Shebang:**
  If the shebang line is omitted, the script will be executed by the shell you are currently using. If you are in a bash shell, it will run as a bash script. However, specifying the interpreter with a shebang ensures that the script runs with the intended shell, regardless of the user's current shell.

Variables=Bash

In Bash, a variable is a named storage location that holds a value. Variables allow you to store data, such as strings, numbers, or even the output of commands, and use that data in your scripts or command-line operations.

Defining Variables

1. **Creating a Variable:**
   To create a variable, simply assign a value to it without spaces around the `=` sign.

   ```
   my_var="Hello, World!"
   ```

2. **Accessing a Variable:**
   To access the value of a variable, prefix it with a dollar sign `$`.

   ```
   echo $my_var
   ```

Examples

1. **String Variable:**

   ```
   greeting="Hello"
   name="Alice"
   echo "$greeting, $name!"  # Output: Hello, Alice!
   ```

2. **Numeric Variable:**

   ```
   num1=10
   num2=20
   sum=$((num1 + num2))  # Arithmetic operation
   echo "Sum: $sum"      # Output: Sum: 30
   ```

3. **Command Substitution:**
   You can assign the output of a command to a variable using backticks `` ` `` or `$(...)`.

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

4. **Array Variables:**
   Bash supports arrays, which can hold multiple values.

```
fruits=("apple" "banana" "cherry")
echo "First fruit: ${fruits[0]}"  # Output: First fruit: a
pple
```

5. **Environment Variables:**
   You can export a variable to make it available to child processes.

```
export MY_VAR="This is an environment variable"
```

6. **Read-Only Variables:**
   You can make a variable read-only, meaning it cannot be modified.

```
readonly const_var="I cannot be changed"
```

MATHS in Bash

Bash supports basic arithmetic operations and mathematical calculations. You can perform arithmetic in Bash using several methods.

1. Arithmetic Expansion

You can use the `$((...))` syntax for arithmetic expansion, which allows you to perform calculations directly.

Example:

```
num1=10
num2=5
sum=$((num1 + num2))
echo "Sum: $sum"  # Output: Sum: 15
```

2. Basic Arithmetic Operations

You can perform various arithmetic operations using the following symbols:

- Addition: `+`
- Subtraction: ` `
- Multiplication: ` `
- Division: `/`
- Modulus: `%`

Example:

```
a=20
b=3

addition=$((a + b))
subtraction=$((a - b))
multiplication=$((a * b))
division=$((a / b))
modulus=$((a % b))

echo "Addition: $addition"     # Output: Addition: 23
echo "Subtraction: $subtraction" # Output: Subtraction: 17
echo "Multiplication: $multiplication" # Output: Multiplicati
on: 60
echo "Division: $division"      # Output: Division: 6
echo "Modulus: $modulus"        # Output: Modulus: 2
```

3. Using `expr`

You can also use the `expr` command for arithmetic operations. However, it requires spaces around operators.

Example:

```
x=15
y=4

sum=$(expr $x + $y)
diff=$(expr $x - $y)
prod=$(expr $x \\* $y)
quot=$(expr $x / $y)
mod=$(expr $x % $y)

echo "Sum: $sum"    # Output: Sum: 19
echo "Difference: $diff" # Output: Difference: 11
echo "Product: $prod" # Output: Product: 60
echo "Quotient: $quot" # Output: Quotient: 3
echo "Modulus: $mod"   # Output: Modulus: 3
```

4. Floating Point Arithmetic

Bash does not support floating-point arithmetic directly. However, you can use external tools like `bc` (a command-line calculator) for this purpose.

Example:

```
num1=5.5
num2=2.2

result=$(echo "$num1 + $num2" | bc)
echo "Result: $result"  # Output: Result: 7.7
```

5. Using `bc` for More Complex Calculations

You can use `bc` for more complex mathematical operations, including functions like square root, trigonometric functions, etc.

Example:

```
# Calculate square root
num=25
sqrt=$(echo "scale=2; sqrt($num)" | bc)
echo "Square root of $num is: $sqrt"  # Output: Square root o
f 25 is: 5.00

# Using scale to control decimal places
result=$(echo "scale=4; 10 / 3" | bc)
echo "10 divided by 3 is: $result"  # Output: 10 divided by 3
is: 3.3333
```

Read

The `read` command in Bash is used to take input from the user. The `-p` option allows you to specify a prompt that is displayed before the user input is read.

Syntax

```
read -p "Your prompt here: " variable_name
```

Example 1: Simple Prompt

```
#!/bin/bash

# Prompt the user for their name
read -p "Enter your name: " name

# Print a greeting
echo "Hello, $name!"
```

Explanation:

- `p "Enter your name: "`: The `p` option displays the prompt "Enter your name: " and waits for the user to input their name.
- `name`: The input from the user is stored in the variable `name`.

Example 2: Using Multiple Variables

You can use `read` to accept multiple inputs at once:

```
#!/bin/bash

# Prompt the user for their first and last name
read -p "Enter your first and last name: " first_name last_na
me

# Print a greeting
echo "Hello, $first_name $last_name!"
```

Explanation:

- The input is split into the variables `first_name` and `last_name` based on whitespace.

Example 3: Input with Default Value

You can use `read` with a default value by using parameter expansion:

```
#!/bin/bash

# Prompt the user with a default value
read -p "Enter your name [default: John]: " name
name=${name:-John}

# Print a greeting
echo "Hello, $name!"
```

Explanation:

- If the user presses Enter without typing anything, `name` will be set to "John" by default.

Example 4: Silent Input (e.g., Password)

You can use `-s` with `read` to take input silently (useful for passwords):

```
#!/bin/bash

# Prompt the user for a password
read -sp "Enter your password: " password
echo

# Print a confirmation message
echo "Password has been set."
```

Explanation:

- `s`: The `s` option hides the user's input as they type, useful for passwords.

Summary

- `p` is a helpful option for providing a prompt message to the user.
- It allows for interactive scripts that require user input with clear instructions.

if [condition]; then

Conditions in Bash are used to make decisions based on the outcome of a command or a set of conditions. They are crucial for controlling the flow of your scripts.

1. If-Else Statement

The if-else statement is used to execute a block of code if a condition is true, and another block if it's false.

Example: Checking if a file exists

```
#!/bin/bash

file="example.txt"

if [ -f "$file" ]; then
  echo "The file $file exists."
else
  echo "The file $file does not exist."
fi        #This keyword is used to end the statement
```

2. If-Elif-Else Statement

The if-elif-else statement is used to check multiple conditions and execute different blocks of code based on those conditions.

Example: Checking the age of a user

```
#!/bin/bash

age=25

if [ $age -lt 18 ]; then
  echo "You are a minor."
elif [ $age -ge 18 ] && [ $age -lt 65 ]; then
  echo "You are an adult."
else
  echo "You are a senior citizen."
fi
```

3. Case Statement

The case statement is used to execute a block of code based on a pattern.

Example: Checking the day of the week

```
#!/bin/bash
```

```
day=$(date +%A)

case $day in
  Monday)
    echo "Today is Monday"
    ;;
  Tuesday)
    echo "Today is Tuesday"
    ;;
  Wednesday)
    echo "Today is Wednesday"
    ;;
  Thursday)
    echo "Today is Thursday"
    ;;
  Friday)
    echo "Today is Friday"
    ;;
  Saturday)
    echo "Today is Saturday"
    ;;
  Sunday)
    echo "Today is Sunday"
    ;;
  *)
    echo "Invalid day"
    ;;
esac
```

4. Logical Operators

Logical operators are used to combine conditions.

Example: Checking if a user is an adult and a citizen

```
#!/bin/bash

age=25
citizen="yes"

if [ $age -ge 18 ] && [ "$citizen" = "yes" ]; then
  echo "You are an adult citizen."
else
  echo "You are not an adult citizen."
fi
```

5. Arithmetic Expansion

Arithmetic expansion is used to evaluate arithmetic expressions.

Example: Checking if a number is even or odd

```
#!/bin/bash

num=10

if (( $num % 2 == 0 )); then
  echo "The number is even."
else
  echo "The number is odd."
fi
```

6. String Comparison

String comparison is used to compare strings.

Example: Checking if a string contains a substring

```
#!/bin/bash

str="Hello World"
substr="World"
```

```
if [[ $str == *$substr* ]]; then
  echo "The string contains the substring."
else
  echo "The string does not contain the substring."
fi
```

Exit codes

In Bash, when a command or script is executed, it returns an exit code (also known as a return status or exit status). This exit code is a numeric value that indicates the success or failure of the command or script. The exit code can be accessed using the special variable `$?`.

Common Exit Codes

- **0**: Success. The command or script executed without errors.
- **1**: General error. A command failed, but it's not specific to any type of error.
- **2**: Misuse of shell built-ins. For example, using an invalid argument with a built-in command.
- **126**: Command invoked cannot be executed (e.g., permission problem or not an executable).
- **127**: Command not found. This occurs when you try to execute a command that doesn't exist.
- **128**: Invalid argument to `exit`. This code indicates an error in specifying an exit status.
- **130**: Script terminated by Control-C (i.e., it was killed using `Ctrl+C`).
- **255**: Exit status out of range. Exit codes should be between 0 and 255. If a script attempts to exit with a code outside this range, it will return 255.

Using Exit Codes in Scripts

Exit codes are useful in scripting for flow control, allowing you to determine the next steps based on whether previous commands succeeded or failed.

Example: Checking the Exit Code

```
#!/bin/bash

# Try to create a directory
mkdir /tmp/mydir

# Check if the command was successful
if [ $? -eq 0 ]; then
    echo "Directory created successfully."
else
    echo "Failed to create directory."
fi
```

In this example, after attempting to create a directory, the script checks the exit code of the `mkdir` command using `$?`. If the exit code is `0`, the script prints a success message; otherwise, it prints a failure message.

Setting Custom Exit Codes in Scripts

You can set custom exit codes in your Bash scripts using the `exit` command followed by a number.

```
#!/bin/bash

if [ "$1" == "" ]; then
    echo "No arguments provided!"
    exit 1  # Exit with a custom code of 1
fi

echo "Argument provided: $1"
exit 0  # Exit with a success code of 0
```

In this script, if no argument is provided, it exits with a code of `1`, indicating an error. Otherwise, it exits with `0`, indicating success.

Loops

Loops in Bash are used to execute a series of commands repeatedly. They are essential for automating repetitive tasks. Bash supports several types of loops, including `for`, `while`, and `until` loops.

1. `for` Loop

The `for` loop iterates over a list of items and executes a block of code for each item.

Syntax:

```
for variable in list
do
  # Commands to execute
done
```

Example:

```
#!/bin/bash

# Loop through a list of fruits
for fruit in apple banana cherry
do
  echo "I like $fruit"
done
```

Output:

```
I like apple
I like banana
I like cherry
```

2. `while` Loop

The `while` loop repeatedly executes a block of code as long as a given condition is true.

Syntax:

```
while [ condition ]
do
  # Commands to execute
done
```

Example:

```
#!/bin/bash

# Loop until the counter reaches 5
counter=1
while [ $counter -le 5 ]
do
  echo "Counter: $counter"
  ((counter++))
done
```

Output:

```
Counter: 1
Counter: 2
Counter: 3
```

```
Counter: 4
Counter: 5
```

3. `until` Loop

The `until` loop is similar to the `while` loop, but it executes the block of code until the condition becomes true.

Syntax:

```
until [ condition ]
do
  # Commands to execute
done
```

Example:

```
#!/bin/bash

# Loop until the counter reaches 5
counter=1
until [ $counter -gt 5 ]
do
  echo "Counter: $counter"
  ((counter++))
done
```

Output:

```
Counter: 1
Counter: 2
Counter: 3
Counter: 4
Counter: 5
```

4. `break` and `continue` Statements

- `break`: Exits the loop entirely.
- `continue`: Skips the current iteration and moves to the next iteration of the loop.

Example with `break`:

```
#!/bin/bash

for i in {1..10}
do
  if [ $i -eq 5 ]
  then
    break
  fi
  echo "Number: $i"
done
```

Output:

```
Number: 1
Number: 2
Number: 3
Number: 4
```

Example with `continue`:

```
#!/bin/bash

for i in {1..5}
do
  if [ $i -eq 3 ]
  then
    continue
  fi
```

```
  echo "Number: $i"
done
```

Output:

```
Number: 1
Number: 2
Number: 4
Number: 5
```

5. Nested Loops

Loops can be nested, meaning you can have one loop inside another.

Example:

```
#!/bin/bash

for i in 1 2 3
do
  for j in a b c
  do
    echo "i=$i, j=$j"
  done
done
```

Output:

```
i=1, j=a
i=1, j=b
i=1, j=c
i=2, j=a
i=2, j=b
i=2, j=c
i=3, j=a
```

```
i=3, j=b
i=3, j=c
```

("Array")

Arrays in Bash

In Bash, arrays are a way to store multiple values in a single variable. Unlike some programming languages, Bash only supports one-dimensional arrays.

Declaring an Array

You can declare an array in Bash using the following syntax:

```
# Method 1: Declare an array with values
my_array=("value1" "value2" "value3")

# Method 2: Declare an empty array
my_array=()

# Method 3: Declare an array with specific indexes
my_array[0]="value1"
my_array[1]="value2"
my_array[2]="value3"
```

Accessing Array Elements

You can access individual elements in an array using their index:

```
# Access the first element
echo "${my_array[0]}"

# Access the second element
echo "${my_array[1]}"
```

To access all elements in an array:

```
# Access all elements
echo "${my_array[@]}"

# Access all elements with indices
for i in "${!my_array[@]}"; do
    echo "Index $i: ${my_array[$i]}"
done
```

Modifying Array Elements

You can modify an existing element in an array by assigning a new value to the desired index:

```
# Modify the second element
my_array[1]="new_value"
```

Adding Elements to an Array

You can add elements to an array by simply assigning a value to a new index:

```
# Add a new element
my_array[3]="value4"
```

Removing Elements from an Array

You can remove elements from an array using the `unset` command:

```
# Remove the second element
unset my_array[1]
```

Length of an Array

To get the length (number of elements) of an array:

```
# Length of the array
echo "Length of array: ${#my_array[@]}"
```

Example: Looping Through an Array

Here's an example of looping through an array:

```
#!/bin/bash

# Declare an array
fruits=("apple" "banana" "cherry")

# Loop through array elements
for fruit in "${fruits[@]}"; do
    echo "I like $fruit"
done
```

Use Case Example: Script Using Arrays

Suppose you want to create a script that stores a list of services and checks if they are running:

```
#!/bin/bash

# Array of services
services=("nginx" "mysql" "ssh")

# Loop through each service and check its status
for service in "${services[@]}"; do
    if systemctl is-active --quiet "$service"; then
        echo "$service is running"
    else
        echo "$service is not running"
    fi
done
```

This script will check the status of `nginx`, `mysql`, and `ssh` services and print whether each is running.

Functions( ){ }

Functions in Bash scripting are used to group a set of commands that can be executed together. They help in organizing code, improving readability, and reusing code.

Defining a Function

To define a function in Bash, use the following syntax:

```
function_name() {
    # commands
}
```

Or, alternatively:

```
function function_name {
    # commands
}
```

Example Function

Here's a simple example of a Bash function:

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

In this example, `greet` is a function that takes one argument (`$1`) and prints a greeting message.

Calling a Function

To call a function, simply use its name followed by any required arguments:

```
greet "Alice"
```

Output:

```
Hello, Alice!
```

Using Return Values

Bash functions can return values using the `return` statement, but this only works for integer values (exit status). For more complex return values, you can use `echo` and capture the output.

Return Status Example:

```
is_even() {
    if [ $(( $1 % 2 )) -eq 0 ]; then
        return 0  # success
    else
        return 1  # failure
    fi
}

is_even 4
if [ $? -eq 0 ]; then
    echo "The number is even."
else
    echo "The number is odd."
fi
```

Output Example:

```
The number is even.
```

Returning Values with `echo`:

```
get_sum() {
    echo $(( $1 + $2 ))
}

sum=$(get_sum 5 7)
echo "The sum is $sum"
```

Output Example:

```
The sum is 12
```

Function Parameters

Functions can accept parameters and use them inside the function body. Parameters are accessed using `$1`, `$2`, etc., where `$1` is the first argument, `$2` is the second, and so on. `$@` refers to all arguments, and `$#` refers to the number of arguments.

Example Function with Parameters:

```
calculate_area() {
    local width=$1
    local height=$2
    local area=$(( width * height ))
    echo "The area is $area"
}

calculate_area 5 10
```

Output Example:

```
The area is 50
```

Local Variables

Variables inside functions can be declared as `local` to limit their scope to the function. This prevents them from affecting or being affected by variables outside the function.

```
increment() {
    local value=$1
    local result=$(( value + 1 ))
    echo $result
}

num=5
incremented_value=$(increment $num)
echo "Incremented value is $incremented_value"
```

Output Example:

```
Incremented value is 6
```

Function Example with Error Handling

```
divide() {
    if [ $2 -eq 0 ]; then
        echo "Error: Division by zero"
        return 1
    else
        echo $(( $1 / $2 ))
        return 0
    fi
}

result=$(divide 10 2)
if [ $? -eq 0 ]; then
    echo "Result: $result"
else
```

```
    echo "Division failed"
fi
```

Output Example:

```
Result: 5
```

Jobs

Scheduling jobs in Linux can be done using a variety of tools, depending on how you want to schedule the tasks. The most commonly used tools are `cron`, `at`, and `systemd timers`.

1. Scheduling Jobs with `cron`

`cron` is a time-based job scheduler in Unix-like operating systems. You can schedule scripts or commands to run periodically at fixed times, dates, or intervals.

Creating a `Cron` Job

To create a cron job, you edit the crontab (cron table) for a specific user. Each user has their own crontab.

Steps:

1. **Open the Crontab File**:

   ```
   crontab -e
   ```

   This opens the crontab file in the default text editor.

2. **Add a Cron Job**:

   The syntax for a cron job is:

   ```
   * * * * * command_to_run
   ```

   The five asterisks represent:

Field	Description	Allowed Values
Minute	Minute of the hour	0-59
Hour	Hour of the day	0-23
Day of Month	Day of the month	1-31
Month	Month of the year	1-12
Day of Week	Day of the week (Sunday = 0)	0-7 (0 and 7 are Sunday)

Example:

```
0 2 * * * /path/to/script.sh
```

This runs the script at 2:00 AM every day.

3. **Save and Exit**:

   After adding your job, save the file and exit the editor. The job will be automatically scheduled.

Common Examples:

- Run a script every day at midnight:

```
0 0 * * * /path/to/your_script.sh
```

- Run a script every 5 minutes:

```
*/5 * * * * /path/to/your_script.sh
```

- Run a script every Monday at 8:00 AM:

```
0 8 * * 1 /path/to/your_script.sh
```

2. Scheduling One-Time Jobs with `at`

`at` is used for scheduling one-time jobs. Unlike `cron`, which is used for recurring tasks, `at` runs a command or script at a specific time only once.

Using `at`:

1. **Install `at` (if not already installed)**:

   ```
   sudo apt-get install at
   ```

2. **Schedule a Job**:

   ```
   echo "/path/to/script.sh" | at 2:30 PM
   ```

   This will run the script at 2:30 PM today.

   Alternatively, you can enter the command interactively:

   ```
   at 2:30 PM
   ```

   Then type the command you want to run:

   ```
   /path/to/script.sh
   ```

   Press `Ctrl + D` to finish.

3. **List Scheduled `at` Jobs**:

   ```
   atq
   ```

4. **Remove a Scheduled `at` Job**:

   ```
   atrm <job_number>
   ```

3. Scheduling with `systemd` Timers

If your system uses `systemd`, you can use `systemd` timers as a more modern and powerful way to schedule jobs. `systemd` timers can be used to replace cron jobs, offering more complex scheduling options and better integration with other `systemd` features.

Using `systemd` Timers:

1. **Create a Timer Unit File**:

   Timer unit files are stored in `/etc/systemd/system/`. Create a `.timer` file:

   ```
   sudo nano /etc/systemd/system/mytask.timer
   ```

   Example Content:

   ```
   [Unit]
   Description=Run my task

   [Timer]
   OnCalendar=*-*-* 02:00:00
   Persistent=true

   [Install]
   WantedBy=timers.target
   ```

   This schedules the task to run daily at 2:00 AM.

2. **Create a Service Unit File**:

   The timer triggers a service unit. Create the service unit file:

   ```
   sudo nano /etc/systemd/system/mytask.service
   ```

   Example Content:

   ```
   [Unit]
   Description=Run my task script

   [Service]
   ExecStart=/path/to/script.sh
   ```

3. **Enable and Start the Timer**:

```
sudo systemctl enable mytask.timer
sudo systemctl start mytask.timer
```

4. **Check Timer Status**:

```
systemctl status mytask.timer
```

Summary

- `cron`: For recurring tasks (daily, weekly, etc.).
- `at`: For one-time jobs.
- `systemd timers`: For more complex or integrated scheduling, particularly on systems using `systemd`.

Arguments

Arguments in Bash Scripting

Arguments in Bash scripting allow you to pass data or values to your script, making it more flexible and reusable. Here's a detailed explanation of how to use them, along with some practical examples.

1. Positional Parameters

- Positional parameters are used to refer to the arguments passed to a script. They are denoted by `$1`, `$2`, `$3`, and so on. `$0` refers to the script's name.

  Example:

```
#!/bin/bash
echo "First argument: $1"
echo "Second argument: $2"
echo "Script name: $0"
```

  Usage:

```
./script.sh arg1 arg2
```

Output:

```
First argument: arg1
Second argument: arg2
Script name: ./script.sh
```

2. Special Parameters

- `$#`: The number of arguments passed to the script.
- `$*`: All the arguments passed to the script as a single string.
- `$@`: All the arguments passed to the script, but each argument is treated as a separate string.
- `$$`: The process ID of the script.
- `$?`: The exit status of the last executed command.
- `$!`: The process ID of the last background job.

Example:

```
#!/bin/bash
echo "Number of arguments: $#"
echo "All arguments as a single string: $*"
echo "All arguments as separate strings: $@"
echo "Process ID of the script: $$"
echo "Exit status of the last command: $?"
```

3. Shift Command

- The `shift` command is used to shift the positional parameters to the left. This means `$2` becomes `$1`, `$3` becomes `$2`, and so on. This is useful when you need to process each argument in a loop.

Example:

```
#!/bin/bash
while [[ $# -gt 0 ]]; do
  echo "Processing argument: $1"
  shift
done
```

Usage:

```
./script.sh arg1 arg2 arg3
```

Output:

```
Processing argument: arg1
Processing argument: arg2
Processing argument: arg3
```

4. Using Arguments in Functions

- You can also pass arguments to functions within a script. Function arguments are accessed using `$1`, `$2`, etc., similar to script arguments.

  Example:

```
#!/bin/bash
my_function() {
  echo "First argument: $1"
  echo "Second argument: $2"
}

my_function arg1 arg2
```

  Output:

```
First argument: arg1
Second argument: arg2
```

5. Practical Use Case

- Consider a script that backs up files. The source directory and the destination directory can be passed as arguments.

  Example:

```
#!/bin/bash
SOURCE_DIR=$1
DEST_DIR=$2

if [ -d "$SOURCE_DIR" ] && [ -d "$DEST_DIR" ]; then
  cp -r "$SOURCE_DIR"/* "$DEST_DIR"
  echo "Backup completed from $SOURCE_DIR to $DEST_DIR"
else
  echo "Either source or destination directory doesn't exi
st."
fi
```

  Usage:

```
./backup.sh /path/to/source /path/to/destination
```

  Output:

```
Backup completed from /path/to/source to /path/to/destinat
ion
```