Rashtreeya Sikshana Samithi Trust

RV UNIVERSITY

School of Computer Science and Engineering

Bengaluru – 560059

INTRODUCTION TO OPERATING SYSTEMS

CODE: CS1181

II SEMESTER BCA (Hons.)

Introduction to Operating Systems

LAB INSTRUCTORS' MANUAL

2024-2025

Prepared by: Dr. Manish Kumar

Verified by: Program Director, SoCSE

Vision and Mission of the School of Computer Science and Engineering

Vision

To be a pioneering school of Computer Science and Engineering committed to fostering liberal education and empowering the next generation of technologists to make a positive global socio-economic impact.

Mission

1. To be a pioneer in computer science education and benchmark ourselves with the world's top computer science and engineering institutions.
2. To provide state-of-the-art facilities that enable exemplary pedagogy, advanced research, innovation and entrepreneurship in emerging technologies of computer science.
3. To promote a culture of cooperation and inclusiveness among students and faculty from diverse communities enabling them to take part in interdisciplinary and multidisciplinary research, contributing to institution-building.
4. To foster excellence through national and international academic, industry collaborations, bringing in diverse perspectives to drive innovation.
5. To nurture a talented pool of ethical, self-driven and empathetic problem solvers to achieve sustainable development goals.

PROGRAM EDUCATIONAL OBJECTIVES (PEOs)

PEO1: Graduates will demonstrate proficiency in solving industrial and societal problems, conducting research, innovation and entrepreneurship.

PEO2: Graduates will demonstrate strong leadership by excelling in interdisciplinary and multidisciplinary teamwork for sustainable global development.

PEO3: Graduates will embrace lifelong learning and digital proficiency with ethical standards.

PROGRAM OUTCOMES (POs)

Engineering Graduates will be able to:

PO1: **Complex Problem-Solving and Critical Thinking:** Graduates should be proficient in using software tools to solve diverse problems in both familiar and unfamiliar contexts. They should apply their learning to real-life situations through software applications, demonstrating the ability to utilize analytic thought in processing a body of knowledge. This includes the use of software for the analysis and evaluation of policies, practices, evidence, and arguments, as well as assessing the reliability and relevance of evidence. Graduates should be adept at using software to identify assumptions and implications, formulate coherent arguments, detect logical flaws in others' arguments, and synthesize data from various sources. Ultimately, they should be able to draw valid conclusions supported by evidence and examples through the effective use of software tools

PO2: **Creative and Effective Communication:** Description: Graduates should be able to demonstrate creativity by thinking in diverse ways, dealing with complex problems, innovating, and viewing situations from multiple perspectives. They should also effectively communicate by listening carefully, presenting complex information clearly, expressing thoughts and ideas both in writing and orally, sharing views confidently, constructing logical arguments using correct technical language, and conveying ideas respectfully and sensitively to different audiences.

PO3: **Analytical and Research Competence**: Graduates should be able to evaluate the reliability and relevance of evidence, identify logical flaws in arguments, and analyze and synthesize data from various sources to draw valid conclusions and support them with evidence. They should also demonstrate research-related skills, including a keen sense of observation and inquiry, the ability to define problems and formulate relevant research questions, design research proposals, use appropriate methodologies and tools, and apply statistical and analytical techniques.

PO4: **Teamwork and Leadership**: Graduates should be able to work effectively and respectfully with diverse teams, facilitating cooperative efforts and acting together towards common goals. They should demonstrate leadership by mapping out tasks, setting directions, formulating inspiring visions, building and motivating teams to achieve these visions, and using management skills to guide team members to the right destination.

PO5: **Lifelong Learning and Digital Proficiency**: Graduates should demonstrate the ability to acquire new knowledge and skills for lifelong learning, adapt to changing workplace demands, work independently, and manage time and resources effectively. They should also exhibit digital and technological proficiency by using ICT in various learning and work situations, accessing and evaluating information sources, and using appropriate software for data analysis.

PO6: **Multicultural Competence and Ethical Values**: Graduates should demonstrate knowledge of multiple cultures, interact respectfully with diverse groups, lead diverse teams, and adopt a gender-neutral and empathetic approach. They should also embrace and practice constitutional, humanistic, ethical, and moral values, engage in responsible global citizenship, recognize and address ethical issues, follow ethical practices, and promote sustainability and integrity in all aspects of their work.

PO7: **Responsible Autonomy and Environmental Sustainability**: Graduates should demonstrate the ability to apply knowledge and skills independently, manage projects to completion, exercise responsibility, and ensure workplace safety and security. They should also be able to take appropriate actions to mitigate environmental degradation, climate change, and pollution, practice effective waste management, conserve biodiversity, and promote sustainable development and living.

PO8: **Community Engagement and Empathy**: Graduates should demonstrate the capability to participate in community-engaged services and activities to promote societal well-being, and the ability to empathize by understanding the perspectives, experiences, and emotions of others.

Course Outcomes: After completing the course, the students will be able to:	
CO1	Explain the fundamentals of operating system
CO2	Describe device management and input/output handling
CO3	Illustrate different conditions for deadlock and their possible solutions
CO4	Analyze process management and its scheduling techniques
CO5	Analyze memory management and its allocation policies

CO-PO Mapping												
CO/PO	PO1	PO2	PO3	PO4	PO5	PO6	PO7	PO8	PO9	PO10	PO11	PO12
CO1	3	-	-	-	-	3	-	-	-	-	-	-
CO2	-	3	-	2	-	-	-	-	-	-	-	-
CO3	-	-	-	-	-	3	-	-	3	-	-	-
CO4	-	3	-	3	-	-	-	-	-	-	-	-
CO5	-	-	-	-	3	1	-	-	1	-	-	-

3: High, 2: Medium, 1: Low

LIST OF PROGRAMS

PART – A

Sl. No.	Program
1.	**Process Creation with fork()**: Write a shell script that creates child processes using the `fork()` system call. Use `ps` to display the parent and child processes along with their IDs.
2.	**Process Control**: Create a shell script that demonstrates process control commands like `bg`, `fg`, `jobs`, and `kill`. Implement a simple background task and manage its execution.
3.	**Basic Shell Script for Job Scheduling**: Develop a shell script that takes a list of commands and executes them with a specified delay using the `sleep` command. Demonstrate job scheduling in the shell.
4.	**File Operations**: Create a shell script that performs file operations, such as creating, copying, moving, and deleting files using commands like `touch`, `cp`, `mv`, and `rm`. Include error handling to manage file operations gracefully.
5.	**File Permissions**: Write a shell script to modify file permissions and ownership using `chmod` and `chown`. Include user input for the filename and desired permissions, and display the new file permissions.
6.	**Directory Management**: Develop a shell script that creates, lists, and deletes directories. Use commands like `mkdir`, `ls`, and `rmdir`, and include user prompts for directory names.
7.	**Dynamic Memory Check**: Write a shell script that monitors memory usage on the system using commands like `free` and `top`. Display the current memory usage and available memory.
8.	**Environment Variables**: Create a shell script that sets, exports, and displays environment variables. Use `printenv` and `env` to show the current environment settings.
9.	**Standard I/O Redirection**: Write a shell script that demonstrates input and output redirection. Create a script that reads from a file and writes output to another file using redirection operators (>, >>, <).

10.	**Piping Commands**: Develop a shell script that uses pipes to connect multiple commands. For example, use `grep` to filter output from `ls` and `sort` the results.
11.	**Handling File Descriptors**: Create a shell script that redirects standard output and error to a file. Use commands like `command > output.txt 2> error.txt` to demonstrate file descriptor manipulation.
12.	**Signal Handling**: Write a shell script that traps signals (like `SIGINT`) and executes a custom handler. Use the `trap` command to manage signals and demonstrate their effect on process termination.

SOLUTIONS TO LAB PROGRAMS WITH EXPECTED INPUT AND OUTPUT

1. Process Creation in Unix Shell: Creating Child Processes and Displaying Parent-Child Relationships Using ps

In Unix shell scripting, we can simulate process creation by running commands in the background, which essentially creates child processes. Here's a shell script that creates child processes, displays their process IDs, and shows the parent-child relationship.

```
#!/bin/bash

# Display the parent process ID (PID)
echo "Parent Process ID: $$"

# Function to create a child process
create_child_process() {
    echo "Starting Child Process..."
    sleep 3  # Simulate some work
    echo "Child Process ID: $PPID , Parent Process ID: $$"
}

# Create first child process
create_child_process &

# Create second child process
create_child_process &

# Wait for all child processes to complete
wait

# Display the parent process ID again after child processes complete
echo "All child processes have completed. Back to Parent Process ID: $$"
```

Explanation

- `$$`: Represents the process ID of the parent (main script).
- `$PPID`: Represents the parent process ID for each child process, which will match the main script's process ID.
- `(...) &` or `function_name &`: Runs a function in the background, creating a child process.
- `wait`: Ensures the script waits until all child processes have finished.

Steps to Run

1. Save the script as `unix_process_creation.sh`.
2. Make it executable:

chmod +x unix_process_creation.sh

Run the script:

./unix_process_creation.sh

Viewing Processes with `ps`

While the script is running, you can use the `ps` command to view the processes created:

ps -f | grep unix_process_creation.sh

This will display the main script's process (the parent) and any child processes created.

Sample Output

Parent Process ID: 12345

Starting Child Process...

Starting Child Process...

Child Process ID: 12346, Parent Process ID: 12345

Child Process ID: 12347, Parent Process ID: 12345

All child processes have completed. Back to Parent Process ID: 12345

Explanation of Output

1. **Parent Process ID**: The script starts by displaying its own process ID, which is the parent for the child processes.
2. **Starting Child Process**: Each child process starts and outputs that it's starting.
3. **Child Process ID**: Each child process outputs its own process ID (`$$`) and the parent process ID (`$PPID`), which matches the parent script's ID.
4. **Completion Message**: After both child processes complete, the parent script outputs that all child processes have finished.

You can observe these processes with the `ps` command as well. For example:

ps -f | grep unix_process_creation.sh

This command may output something similar to:

```
UID        PID  PPID  C STIME TTY          TIME CMD
user     12345     1  0 10:00 pts/0    00:00:00 /bin/bash ./unix_process_creation.sh
user     12346 12345  0 10:00 pts/0    00:00:00 /bin/bash ./unix_process_creation.sh (child process)
user     12347 12345  0 10:00 pts/0    00:00:00 /bin/bash ./unix_process_creation.sh (child process)
```

In this example, `12345` is the parent process ID (script), and `12346` and `12347` are the child processes created by it.

2. Process Control: Create a shell script that demonstrates process control commands like `bg`, `fg`, `jobs`, and `kill`. Implement a simple background task and manage its execution.

Demonstrating Process Control Commands

Save the following script as `process_control_demo.sh`.

```
#!/bin/bash

# Start a background task (using sleep to simulate a long-running task)
echo "Starting a background task..."
sleep 300 &  # Run for 5 minutes in the background
task_pid=$!  # Capture the PID of the background task
echo "Background task started with PID: $task_pid"

# Display the jobs running in the background
echo -e "\nListing jobs:"
jobs

# Check if the process is running using ps
echo -e "\nChecking if the task is running..."
ps -p $task_pid

# Pausing the background task
echo -e "\nPausing the background task..."
kill -STOP $task_pid

# Verifying if the task is paused
echo -e "\nChecking task status after pausing (should be stopped):"
ps -o pid,stat,cmd -p $task_pid  # Shows the PID, status (should be T), and command

# Resuming the background task
echo -e "\nResuming the background task..."
kill -CONT $task_pid

# Verifying if the task is resumed
echo -e "\nChecking task status after resuming (should be running):"
ps -o pid,stat,cmd -p $task_pid  # Shows the PID, status (should be R), and command

# Killing the background task
echo -e "\nKilling the background task..."
kill $task_pid

# Confirming the task has been terminated
echo -e "\nChecking task status after termination (should not exist):"
ps -p $task_pid && echo "Process still exists." || echo "Process terminated."
```

Explanation of Each Command

1. **Start a Background Task:**
   - `sleep 300 &`: Runs a `sleep` command in the background for 5 minutes (300 seconds).
   - `task_pid=$!`: Captures the process ID (PID) of the background task, which allows us to control it later in the script.

2. **List Background Jobs:**
   - `jobs`: Lists all background jobs currently running in the shell, showing the job ID, status, and command. This helps us verify that the `sleep` command is running in the background.
3. **Pause the Background Task:**
   - `kill -STOP $task_pid`: Sends the `STOP` signal to the background task using its PID. This pauses the process without terminating it, simulating a "suspend" operation.
4. **List Paused Jobs:**
   - `jobs`: Shows the current status of jobs after the `STOP` signal is sent. The job status should now show as "Stopped."
5. **Resume the Background Task:**
   - `kill -CONT $task_pid`: Sends the `CONT` (continue) signal to the previously paused process, resuming its execution in the background.
6. **List Resumed Jobs:**
   - `jobs`: Confirms that the background task has resumed, showing its status as "Running."
7. **Kill the Background Task:**
   - `kill $task_pid`: Terminates the background process by sending the `TERM` signal. This is the default action of `kill`, which stops the process gracefully.
8. **Confirm Task Termination:**
   - `jobs`: Displays the jobs list one last time to confirm that the background task is no longer running, typically showing the status as "Terminated."

Steps to Run the Script

1. Save the script as `process_control_demo.sh`.
2. Make it executable:

   ```
   chmod +x process_control_demo.sh
   ```

3. Run the script:

   ```
   ./process_control_demo.sh
   ```

Expected Sample Output

```
Starting a background task...
Background task started with PID: 15898

Listing jobs:
[1]+  Running                 sleep 300 &

Checking if the task is running...
  PID TTY          TIME CMD
15898 pts/0    00:00:00 sleep

Pausing the background task...

Checking task status after pausing (should be stopped):
  PID STAT CMD
15898 T    sleep 300

Resuming the background task...

Checking task status after resuming (should be running):
  PID STAT CMD
```

```
15898 S    sleep 300

Terminating the background task...

Checking task status after termination (should not exist):
Process terminated.
```

Explanation of Each Section in the Output

1. **Starting a Background Task**:
   - `Starting a background task...`
   - `Background task started with PID: 15898`
   - This part confirms that the `sleep 300` command was successfully started in the background, with a process ID (PID) of `15898`.
2. **Listing Jobs**:
   - `[1]+ Running sleep 300 &`
   - This output from the `jobs` command confirms that `sleep 300` is running in the background as job `1`, and it is currently active (`Running`).
3. **Checking if the Task is Running**:
   - `PID TTY TIME CMD`
   - `15898 pts/0 00:00:00 sleep`
   - This output from `ps -p $task_pid` shows that the `sleep` process with PID `15898` is running. The `CMD` column shows it's running the `sleep` command, and the `TIME` is `00:00:00` as it has just started.
4. **Pausing the Background Task**:
   - This section does not produce an immediate output but sends a `STOP` signal to pause the task.
5. **Checking Task Status After Pausing**:
   - `PID STAT CMD`
   - `15898 T sleep 300`
   - Here, `ps -o pid,stat,cmd -p $task_pid` shows the process's `STAT` as `T`, indicating it is stopped (or "paused"). The `T` status confirms that the `sleep` command is no longer executing but is paused.
6. **Resuming the Background Task**:
   - This command resumes the task but does not produce immediate output.
7. **Checking Task Status After Resuming**:
   - `PID STAT CMD`
   - `15898 S sleep 300`
   - After resuming, the `STAT` column shows `S`, indicating that the process is in a "sleeping" state and is running in the background. This confirms that the `sleep` command has resumed execution.
8. **Terminating the Background Task**:
   - This command sends a termination signal but does not produce immediate output.
9. **Checking Task Status After Termination**:
   - `Process terminated.`
   - The final check uses `ps -p $task_pid` to verify that the process no longer exists. Since the command produces no output, it indicates the process has been terminated, confirmed by the `Process terminated.` message.

This output and explanation show how the script demonstrates process control commands (`jobs`, `kill -STOP`, `kill -CONT`, and `kill`) and how each state is verified using `ps`.

3. Basic Shell Script for Job Scheduling: Develop a shell script that takes a list of commands and executes them with a specified delay using the sleep command. Demonstrate job scheduling in the shell.

Shell Script

```
#!/bin/bash

# Basic Shell Script for Job Scheduling
# Usage: ./job_scheduler.sh <delay_in_seconds> "command1" "command2" ...

# Check if sufficient arguments are provided
if [ "$#" -lt 2 ]; then
    echo "Usage: $0 <delay_in_seconds> \"command1\" \"command2\" ..."
    exit 1
fi

# Get the delay from the first argument
delay=$1
shift  # Remove the first argument so we can loop over commands

# Execute each command with the specified delay
for command in "$@"; do
    echo "Executing: $command"
    eval "$command"  # Execute the command
    echo "Waiting for $delay seconds..."
    sleep "$delay"  # Delay before executing the next command
done

echo "All commands executed."
```

Explanation

- **Purpose**: This script schedules commands to be executed one after another with a specified delay in between. The user specifies the delay and the commands to be executed.
- **Input**: The first argument is the delay in seconds, and the subsequent arguments are the commands to be executed.
- **Loop**: It iterates through each command, executes it using `eval`, and then waits for the specified delay before proceeding to the next command.

Steps to Run

1. **Create the Script File**:
    - Open a terminal.
    - Use a text editor (like `nano` or `vi`) to create a file named `job_scheduler.sh`:

      nano job_scheduler.sh

2. **Copy and Paste the Script:**

- Copy the script provided above into the editor and save the file.

3. **Make the Script Executable:**

- Change the permission to make the script executable:

  chmod +x job_scheduler.sh

4. Run the Script**:**

- Execute the script by providing a delay and commands:

  ./job_scheduler.sh 5 "echo 'Hello, World!'" "date" "ls -l"

Sample Output

When running the script with the provided example, you might see output similar to the following:

```
Executing: echo Hello, World!
Hello, World!
Waiting for 5 seconds...
Executing: date
Thu Nov  1 12:34:56 UTC 2024
Waiting for 5 seconds...
Executing: ls
Desktop  Documents  Downloads  Music  Pictures  Videos
Waiting for 5 seconds...
All commands executed.
```

Explanation of Output

1. **Executing Commands**:
   - For each command, the script first prints "Executing: [command]" to indicate which command is currently being run.
2. **Command Output**:
   - The result of the executed command is displayed immediately below the execution message. For example, `Hello, World!` appears after the `echo` command is executed.
3. **Delay Message**:
   - After executing each command, the script prints "Waiting for [delay] seconds..." to indicate the pause before the next command execution.
4. **Completion Message**:
   - After all commands have been executed and delayed accordingly, the script outputs "All commands executed." to signify the end of the script's operation.

4. File Operations: Create a shell script that performs file operations, such as creating, copying, moving, and deleting files using commands like touch, cp, mv, and rm. Include error handling to manage file operations gracefully.

Shell Script

```
#!/bin/bash

# Basic Shell Script for File Operations
# Usage: ./file_operations.sh <operation> <source_file> [destination_file]

# Function to display usage information
usage() {
    echo "Usage: $0 {create|copy|move|delete} <source_file> [destination_file]"
    exit 1
}

# Check if at least two arguments are provided
if [ "$#" -lt 2 ]; then
    usage
fi

operation=$1
source_file=$2
destination_file=$3

case "$operation" in
    create)
        # Create a file
        touch "$source_file"
        if [ $? -eq 0 ]; then
            echo "File '$source_file' created successfully."
        else
            echo "Error: Could not create file '$source_file'."
        fi
        ;;
    copy)
        # Copy a file
        if [ -f "$source_file" ]; then
            cp "$source_file" "$destination_file"
            if [ $? -eq 0 ]; then
                echo "File '$source_file' copied to '$destination_file'
successfully."
            else
                echo "Error: Could not copy file '$source_file' to
'$destination_file'."
            fi
        else
            echo "Error: Source file '$source_file' does not exist."
        fi
        ;;
    move)
        # Move (or rename) a file
        if [ -f "$source_file" ]; then
            mv "$source_file" "$destination_file"
            if [ $? -eq 0 ]; then
```

```
                echo "File '$source_file' moved to '$destination_file'
successfully."
            else
                echo "Error: Could not move file '$source_file' to
'$destination_file'."
            fi
        else
            echo "Error: Source file '$source_file' does not exist."
        fi
        ;;
    delete)
        # Delete a file
        if [ -f "$source_file" ]; then
            rm "$source_file"
            if [ $? -eq 0 ]; then
                echo "File '$source_file' deleted successfully."
            else
                echo "Error: Could not delete file '$source_file'."
            fi
        else
            echo "Error: File '$source_file' does not exist."
        fi
        ;;
    *)
        usage
        ;;
esac
```

Explanation

- **Purpose**: This script performs basic file operations: creating, copying, moving, and deleting files. It also includes error handling to manage potential issues during file operations.
- **Input**: The first argument specifies the operation (`create`, `copy`, `move`, `delete`), the second argument is the source file, and the third argument (optional) is the destination file (used in `copy` and `move` operations).
- **Error Handling**: After each operation, the script checks the exit status (`$?`) to determine if the operation was successful or if an error occurred, providing appropriate feedback to the user.

Steps to Run

1. **Create the Script File**:
   - Open a terminal.
   - Use a text editor (like `nano` or `vi`) to create a file named `file_operations.sh`:

     ```
     bash
     Copy code
     nano file_operations.sh
     ```

2. **Copy and Paste the Script**:
   - Copy the script provided above into the editor and save the file.
3. **Make the Script Executable**:
   - Change the permission to make the script executable:

     ```
     bash
     Copy code
     chmod +x file_operations.sh
     ```

4. **Run the Script**:
    - Execute the script by providing an operation and the required arguments. Here are some examples:
        - Create a file:

```
./file_operations.sh create myfile.txt
```

        - Copy a file:

```
./file_operations.sh copy myfile.txt myfile_copy.txt
```

        - Move (rename) a file:

```
./file_operations.sh move myfile_copy.txt myfile_renamed.txt
```

        - Delete a file:

```
./file_operations.sh delete myfile_renamed.txt
```

Sample Output

When you run the commands, you might see output similar to the following, depending on the operation performed:

```
# Create a file
$ ./file_operations.sh create myfile.txt
File 'myfile.txt' created successfully.

# Copy a file
$ ./file_operations.sh copy myfile.txt myfile_copy.txt
File 'myfile.txt' copied to 'myfile_copy.txt' successfully.

# Move (rename) a file
$ ./file_operations.sh move myfile_copy.txt myfile_renamed.txt
File 'myfile_copy.txt' moved to 'myfile_renamed.txt' successfully.

# Delete a file
$ ./file_operations.sh delete myfile_renamed.txt
File 'myfile_renamed.txt' deleted successfully.
```

Explanation of Output

1. **File Creation**:
    - After executing the create command, it confirms that the file was created successfully.
2. **File Copying**:
    - When copying a file, it checks for the existence of the source file and confirms the copy operation.
3. **File Moving**:
    - The move operation renames or relocates the file, and it also checks if the source file exists before proceeding.
4. **File Deletion**:
    - The script confirms deletion after checking that the file exists.
5. **Error Handling**:

- If any operation fails (e.g., trying to copy a non-existent file), the script provides an error message explaining the issue.

This shell script effectively demonstrates file operations with basic error handling to ensure smooth execution and user feedback.

5. File Permissions: Write a shell script to modify file permissions and ownership using chmod and chown. Include user input for the filename and desired permissions, and display the new file permissions.

Shell Script

```
#!/bin/bash

# Basic Shell Script for Modifying File Permissions and Ownership
# Usage: ./file_permissions.sh <filename>

# Function to display usage information
usage() {
    echo "Usage: $0 <filename>"
    exit 1
}

# Check if a filename is provided
if [ "$#" -ne 1 ]; then
    usage
fi

filename=$1

# Check if the file exists
if [ ! -e "$filename" ]; then
    echo "Error: File '$filename' does not exist."
    exit 1
fi

# Display current permissions
current_permissions=$(ls -l "$filename" | awk '{print $1}')
echo "Current permissions for '$filename': $current_permissions"

# Get desired permissions from user
read -p "Enter new permissions (e.g., 755 for rwxr-xr-x): " new_permissions

# Change file permissions
chmod "$new_permissions" "$filename"
if [ $? -eq 0 ]; then
    echo "Permissions for '$filename' changed to $new_permissions."
else
    echo "Error: Could not change permissions for '$filename'."
    exit 1
fi

# Display new permissions
new_permissions_display=$(ls -l "$filename" | awk '{print $1}')
echo "New permissions for '$filename': $new_permissions_display"

# Get ownership change input from user
read -p "Do you want to change the ownership of the file? (y/n): " change_ownership
if [[ "$change_ownership" =~ ^[Yy]$ ]]; then
    read -p "Enter new owner (username): " new_owner
    read -p "Enter new group (groupname): " new_group

    # Change file ownership
    chown "$new_owner":"$new_group" "$filename"
```

```
    if [ $? -eq 0 ]; then
        echo "Ownership for '$filename' changed to $new_owner:$new_group."
    else
        echo "Error: Could not change ownership for '$filename'."
        exit 1
    fi
else
    echo "Ownership change skipped."
fi

# Display final permissions and ownership
echo "Final details for '$filename':"
ls -l "$filename"
```

Explanation

- **Purpose**: This script allows users to modify file permissions and ownership for a specified file. It interacts with the user to gather the necessary input for changes.
- **Input Parameters**:
  - The script takes one argument: the filename on which to perform operations.
  - It prompts the user for new permissions and optional ownership details.
- **Key Commands Used**:
  - `ls -l`: To list the file with permissions and ownership details.
  - `chmod`: To change the file permissions.
  - `chown`: To change the file ownership.
- **Error Handling**:
  - The script checks if the specified file exists and whether the permission changes were successful. If any operation fails, an error message is displayed.

Steps to Run

1. **Create the Script File**:
   - Open a terminal.
   - Use a text editor (like `nano` or `vi`) to create a file named `file_permissions.sh`:

     ```
     nano file_permissions.sh
     ```

2. **Copy and Paste the Script**:
   - Copy the script provided above into the editor and save the file.
3. **Make the Script Executable**:
   - Change the permission to make the script executable:

     ```
     chmod +x file_permissions.sh
     ```

4. **Run the Script**:
   - Execute the script by providing a filename. Here's how to run it:

     ```
     ./file_permissions.sh myfile.txt
     ```

5. **Follow the Prompts**:
   - The script will display the current permissions and prompt you to enter new permissions. For example, if you want to set the permissions to `755`, type `755` and press Enter.
   - You will then be asked if you want to change ownership. If you answer 'y', enter the new owner and group.

Sample Output

Here's a sample interaction with the script:

```
plaintext
Copy code
$ ./file_permissions.sh myfile.txt
Current permissions for 'myfile.txt': -rw-r--r--
Enter new permissions (e.g., 755 for rwxr-xr-x): 755
Permissions for 'myfile.txt' changed to 755.
New permissions for 'myfile.txt': -rwxr-xr-x
Do you want to change the ownership of the file? (y/n): y
Enter new owner (username): alice
Enter new group (groupname): staff
Ownership for 'myfile.txt' changed to alice:staff.
Final details for 'myfile.txt':
-rwxr-xr-x 1 alice staff 0 Nov  1 12:00 myfile.txt
```

Explanation of Output

1. **Current Permissions**:
   - The script initially displays the current permissions for the specified file. For example, `-rw-r--r--` indicates read and write permissions for the owner, and read permissions for the group and others.
2. **Changing Permissions**:
   - After entering new permissions, the script confirms the change and displays the updated permissions. For example, `-rwxr-xr-x` means that the owner can read, write, and execute, while the group and others can read and execute.
3. **Changing Ownership**:
   - If the user chooses to change ownership, the script prompts for the new owner and group. After processing this, it confirms the ownership change.
4. **Final Details**:
   - The final listing of the file with `ls -l` shows the updated permissions and ownership details, verifying that both changes were applied successfully.

This script effectively demonstrates how to modify file permissions and ownership in a user-friendly manner, providing clear feedback throughout the process.

6. Directory Management: Develop a shell script that creates, lists, and deletes directories. Use commands like mkdir, ls, and rmdir, and include user prompts for directory names.

Shell Script

```
#!/bin/bash

# Advanced Shell Script for Directory Management
# Usage: ./directory_management.sh

# Function to display usage information
usage() {
    echo "Usage: $0"
    exit 1
}

# Function to create a directory
create_directory() {
    read -p "Enter the name of the new directory: " dir_name
    if [ -d "$dir_name" ]; then
        echo "Error: Directory '$dir_name' already exists."
    else
        mkdir "$dir_name"
        if [ $? -eq 0 ]; then
            echo "Directory '$dir_name' created successfully."
        else
            echo "Error: Could not create directory '$dir_name'."
        fi
    fi
}

# Function to list existing directories
list_directories() {
    echo "Listing directories:"
    ls -d */ 2>/dev/null || echo "No directories found."
}

# Function to delete a directory
delete_directory() {
    read -p "Enter the name of the directory to delete: " dir_name
    if [ ! -d "$dir_name" ]; then
        echo "Error: Directory '$dir_name' does not exist."
    else
        read -p "Are you sure you want to delete '$dir_name'? (y/n): " confirm
        if [[ "$confirm" =~ ^[Yy]$ ]]; then
            rmdir "$dir_name" 2>/dev/null
            if [ $? -eq 0 ]; then
                echo "Directory '$dir_name' deleted successfully."
            else
                echo "Error: Directory '$dir_name' is not empty. Use 'rm -r' to
remove it."
            fi
        else
            echo "Deletion of '$dir_name' canceled."
        fi
    fi
}
```

```
# Function to delete a non-empty directory
delete_non_empty_directory() {
    read -p "Enter the name of the directory to delete (non-empty): " dir_name
    if [ ! -d "$dir_name" ]; then
        echo "Error: Directory '$dir_name' does not exist."
    else
        read -p "Are you sure you want to delete the non-empty directory
'$dir_name'? (y/n): " confirm
        if [[ "$confirm" =~ ^[Yy]$ ]]; then
            rm -r "$dir_name"
            if [ $? -eq 0 ]; then
                echo "Non-empty directory '$dir_name' deleted successfully."
            else
                echo "Error: Could not delete '$dir_name'."
            fi
        else
            echo "Deletion of '$dir_name' canceled."
        fi
    fi
}

# Displaying options to the user
echo "Advanced Directory Management Script"
echo "1. Create a new directory"
echo "2. List existing directories"
echo "3. Delete an empty directory"
echo "4. Delete a non-empty directory"
echo "5. Exit"

# Loop until the user chooses to exit
while true; do
    read -p "Select an option (1-5): " option

    case $option in
        1)  # Create a new directory
            create_directory
            ;;
        2)  # List existing directories
            list_directories
            ;;
        3)  # Delete an empty directory
            delete_directory
            ;;
        4)  # Delete a non-empty directory
            delete_non_empty_directory
            ;;
        5)  # Exit the script
            echo "Exiting the script. Goodbye!"
            exit 0
            ;;
        *)  # Invalid option
            echo "Invalid option. Please select 1-5."
            ;;
    esac
    echo ""  # Print a newline for better readability
done
```

Explanation

- **Purpose**: This enhanced script manages directories with more robust features, including:
    - Checking for existing directories before creation.
    - Options to delete both empty and non-empty directories, with confirmation prompts.

    - Improved error handling to guide the user effectively.
- **Functions**:
    - **create_directory**: Prompts the user for a directory name, checks if it already exists, and creates it if not.
    - **list_directories**: Lists all existing directories or notifies if none are found.
    - **delete_directory**: Deletes an empty directory after confirming the user's intention.
    - **delete_non_empty_directory**: Deletes a non-empty directory after confirming the user's intention.
- **Error Handling**:
    - The script handles errors such as non-existing directories, attempts to delete non-empty directories, and provides feedback on actions taken.

Steps to Run

1. **Create the Script File**:
    - Open a terminal.
    - Use a text editor (like `nano` or `vi`) to create a file named `directory_management.sh`:

      ```
      nano directory_management.sh
      ```

2. **Copy and Paste the Script**:
    - Copy the script provided above into the editor and save the file.
3. **Make the Script Executable**:
    - Change the permission to make the script executable:

      ```
      chmod +x directory_management.sh
      ```

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

      ```
      ./directory_management.sh
      ```

5. **Follow the Prompts**:
    - The script will display a menu with options to create, list, delete directories, or exit. Follow the prompts to perform the desired operations.

Sample Output

Here's a sample interaction with the script:

```
$ ./directory_management.sh
Advanced Directory Management Script
1. Create a new directory
2. List existing directories
3. Delete an empty directory
4. Delete a non-empty directory
5. Exit
Select an option (1-5): 1
Enter the name of the new directory: my_new_directory
Directory 'my_new_directory' created successfully.

Select an option (1-5): 2
Listing directories:
my_new_directory/
another_directory/
```

```
Select an option (1-5): 3
Enter the name of the directory to delete: my_new_directory
Are you sure you want to delete 'my_new_directory'? (y/n): y
Directory 'my_new_directory' deleted successfully.

Select an option (1-5): 4
Enter the name of the directory to delete (non-empty): another_directory
Are you sure you want to delete the non-empty directory 'another_directory'? (y/n):
y
Non-empty directory 'another_directory' deleted successfully.

Select an option (1-5): 5
Exiting the script. Goodbye!
```

Explanation of Output

1. **Initial Menu**:
   - The script displays options for directory management.
2. **Creating a Directory**:
   - When creating a directory, if it already exists, the script will inform the user and prevent duplicate creation.
3. **Listing Directories**:
   - It shows all existing directories. If none exist, it provides a message indicating that.
4. **Deleting an Empty Directory**:
   - If the user selects to delete an empty directory, the script prompts for confirmation. Upon confirmation, it attempts deletion and provides feedback on success or failure.
5. **Deleting a Non-Empty Directory**:
   - The script similarly handles the deletion of non-empty directories, allowing the user to confirm the action before proceeding.
6. **Exiting the Script**:
   - Finally, the user can exit the script cleanly.

7. Dynamic Memory Check: Write a shell script that monitors memory usage on the system using commands like free and top. Display the current memory usage and available memory.

Shell Script

```
#!/bin/bash

# Shell Script for Monitoring Memory Usage
# Usage: ./memory_monitor.sh

# Function to display memory usage
display_memory_usage() {
    echo "Current Memory Usage:"
    free -h
    echo ""
}

# Function to display memory usage in real-time
monitor_memory_usage() {
    echo "Monitoring memory usage in real-time. Press [CTRL+C] to stop."
    # Use top command to show memory usage updates every 2 seconds
    top -o %MEM
}

# Displaying options to the user
echo "Dynamic Memory Monitor"
echo "1. Display current memory usage"
echo "2. Monitor memory usage in real-time"
echo "3. Exit"

# Loop until the user chooses to exit
while true; do
    read -p "Select an option (1-3): " option

    case $option in
        1)  # Display current memory usage
            display_memory_usage
            ;;
        2)  # Monitor memory usage in real-time
            monitor_memory_usage
            ;;
        3)  # Exit the script
            echo "Exiting the memory monitor. Goodbye!"
            exit 0
            ;;
        *)  # Invalid option
            echo "Invalid option. Please select 1-3."
            ;;
    esac
    echo ""  # Print a newline for better readability
done
```

Explanation

- **Purpose**: The script is designed to monitor and display memory usage on the system. It provides a user-friendly interface with options to either display the current memory status or monitor it in real-time.

- **Functions**:
  - **display_memory_usage**: This function utilizes the `free` command to show the current memory usage in a human-readable format (`-h` flag).
  - **monitor_memory_usage**: This function uses the `top` command to display real-time memory usage, sorted by memory percentage (`-o %MEM`). The user can exit this view using `CTRL+C`.
- **Menu Options**:
  - The script presents a simple menu that allows the user to select between displaying memory usage, monitoring it in real-time, or exiting the script.

Steps to Run

1. **Create the Script File**:
   - Open a terminal.
   - Use a text editor (like `nano` or `vi`) to create a file named `memory_monitor.sh`:

     ```
     nano memory_monitor.sh
     ```

2. **Copy and Paste the Script**:
   - Copy the script provided above into the editor and save the file.
3. **Make the Script Executable**:
   - Change the permission to make the script executable:

     ```
     chmod +x memory_monitor.sh
     ```

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

     ```
     ./memory_monitor.sh
     ```

5. **Follow the Prompts**:
   - The script will display a menu. Select options to display current memory usage or monitor memory usage in real-time.

Sample Output

Here's a sample interaction with the script:

```
plaintext
Copy code
$ ./memory_monitor.sh
Dynamic Memory Monitor
1. Display current memory usage
2. Monitor memory usage in real-time
3. Exit
Select an option (1-3): 1
Current Memory Usage:
              total        used        free      shared  buff/cache   available
Mem:           15Gi       6.5Gi       1.8Gi       502Mi       7.2Gi       7.9Gi
Swap:         2.0Gi          0B       2.0Gi

Select an option (1-3): 2
Monitoring memory usage in real-time. Press [CTRL+C] to stop.
top - 15:23:30 up  5 days,  3:22,  1 user,  load average: 0.15, 0.05, 0.01
Tasks: 228 total,   1 running, 227 sleeping,   0 stopped,   0 zombie
%Cpu(s):  1.5 us,  0.5 sy,  0.0 ni, 97.5 id,  0.0 wa,  0.0 hi,  0.5 si,  0.0 st
```

```
MiB Mem :  16384 total,   6750 free,   6500 used,   7134 buff/cache
MiB Swap:  2048 total,   2048 free,      0 used.   7250 avail Mem

Select an option (1-3): 3
Exiting the memory monitor. Goodbye!
```

Explanation of Output

1. **Initial Menu**:
   - The script presents the user with options for monitoring memory.
2. **Displaying Current Memory Usage**:
   - When the user selects option 1, the `free` command outputs a summary of the current memory usage, showing total, used, free, shared, buffer/cache, and available memory in a human-readable format.
3. **Real-Time Monitoring**:
   - When the user selects option 2, the script enters the `top` command interface, displaying live updates of system processes and memory usage. The user can exit this view at any time using `CTRL+C`.
4. **Exiting the Script**:
   - Finally, selecting option 3 allows the user to exit the script cleanly.

This script serves as a practical tool for monitoring system memory usage, providing users with necessary information and real-time updates, enhancing their understanding of system resources.

8. Environment Variables: Create a shell script that sets, exports, and displays environment variables. Use printenv and env to show the current environment settings.

Shell Script

```
#!/bin/bash

# Shell Script for Managing Environment Variables
# Usage: ./env_variables.sh

# Function to display current environment variables
display_environment_variables() {
    echo "Current Environment Variables:"
    printenv
    echo ""
}

# Function to set and export a new environment variable
set_environment_variable() {
    read -p "Enter the name of the environment variable: " var_name
    read -p "Enter the value of the environment variable: " var_value
    export "$var_name=$var_value"
    echo "Environment variable '$var_name' set to '$var_value'."
    echo ""
}

# Function to display all environment variables with env command
show_env_command() {
    echo "Displaying environment variables using 'env':"
    env
    echo ""
}

# Main menu for user interaction
echo "Environment Variable Manager"
echo "1. Display current environment variables using printenv"
echo "2. Set and export a new environment variable"
echo "3. Display environment variables using env"
echo "4. Exit"

# Loop until the user chooses to exit
while true; do
    read -p "Select an option (1-4): " option

    case $option in
        1)  # Display current environment variables
            display_environment_variables
            ;;
        2)  # Set and export a new environment variable
            set_environment_variable
            ;;
        3)  # Show environment variables using env
            show_env_command
            ;;
        4)  # Exit the script
            echo "Exiting the Environment Variable Manager. Goodbye!"
            exit 0
            ;;
        *)  # Invalid option
```

```
            echo "Invalid option. Please select 1-4."
            ;;
    esac
    echo ""  # Print a newline for better readability
done
```

Explanation

- **Purpose**: This script manages environment variables by allowing users to view current variables, set new ones, and display them using different commands.
- **Functions**:
  - **display_environment_variables**: Uses the `printenv` command to list all current environment variables.
  - **set_environment_variable**: Prompts the user for a variable name and value, then sets and exports the variable in the current shell session.
  - **show_env_command**: Utilizes the `env` command to show all current environment variables.
- **Menu Options**:
  - Users can select options to display environment variables, set a new variable, or exit.

Steps to Run

1. **Create the Script File**:
   - Open a terminal.
   - Create a new script file named `env_variables.sh` using a text editor:

     ```
     nano env_variables.sh
     ```

2. **Copy and Paste the Script**:
   - Paste the provided script into the editor and save the file.
3. **Make the Script Executable**:
   - Change the permission to make the script executable:

     ```
     chmod +x env_variables.sh
     ```

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

     ```
     ./env_variables.sh
     ```

5. **Follow the Prompts**:
   - The script will display a menu for the user to select options.

Sample Output

Here's a sample interaction with the script:

```
$ ./env_variables.sh
Environment Variable Manager
1. Display current environment variables using printenv
2. Set and export a new environment variable
3. Display environment variables using env
4. Exit
Select an option (1-4): 1
Current Environment Variables:
```

```
USER=your_username
HOME=/home/your_username
SHELL=/bin/bash
...

Select an option (1-4): 2
Enter the name of the environment variable: MY_VAR
Enter the value of the environment variable: HelloWorld
Environment variable 'MY_VAR' set to 'HelloWorld'.

Select an option (1-4): 1
Current Environment Variables:
USER=your_username
HOME=/home/your_username
SHELL=/bin/bash
MY_VAR=HelloWorld
...

Select an option (1-4): 3
Displaying environment variables using 'env':
USER=your_username
HOME=/home/your_username
SHELL=/bin/bash
MY_VAR=HelloWorld
...

Select an option (1-4): 4
Exiting the Environment Variable Manager. Goodbye!
```

Explanation of Output

1. **Initial Menu**:
   - The script prompts the user to choose between displaying environment variables, setting a new one, or exiting.
2. **Displaying Current Environment Variables**:
   - When option 1 is selected, the `printenv` command outputs the current environment variables and their values.
3. **Setting a New Environment Variable**:
   - Upon selecting option 2, the user is prompted for a variable name and value. The script sets this variable using `export`, making it available in the current session.
4. **Displaying Variables with `env`**:
   - Option 3 uses the `env` command to list environment variables, showing that the new variable has been set.
5. **Exiting the Script**:
   - Selecting option 4 allows the user to exit the script cleanly.

This script serves as a practical tool for managing environment variables, which are essential for configuring the behavior of processes in Unix-like operating systems.

9. Standard I/O Redirection: Write a shell script that demonstrates input and output redirection. Create a script that reads from a file and writes output to another file using redirection operators (>, >>, <).

Shell Script

```
#!/bin/bash

# Shell Script for Managing Environment Variables
# Usage: ./env_variables.sh

# Function to display current environment variables
display_environment_variables() {
    echo "Current Environment Variables:"
    printenv
    echo ""
}

# Function to set and export a new environment variable
set_environment_variable() {
    read -p "Enter the name of the environment variable: " var_name
    read -p "Enter the value of the environment variable: " var_value
    export "$var_name=$var_value"
    echo "Environment variable '$var_name' set to '$var_value'."
    echo ""
}

# Function to display all environment variables with env command
show_env_command() {
    echo "Displaying environment variables using 'env':"
    env
    echo ""
}

# Main menu for user interaction
echo "Environment Variable Manager"
echo "1. Display current environment variables using printenv"
echo "2. Set and export a new environment variable"
echo "3. Display environment variables using env"
echo "4. Exit"

# Loop until the user chooses to exit
while true; do
    read -p "Select an option (1-4): " option

    case $option in
        1)  # Display current environment variables
            display_environment_variables
            ;;
        2)  # Set and export a new environment variable
            set_environment_variable
            ;;
        3)  # Show environment variables using env
            show_env_command
            ;;
        4)  # Exit the script
            echo "Exiting the Environment Variable Manager. Goodbye!"
            exit 0
```

```
            ;;
        *)  # Invalid option
            echo "Invalid option. Please select 1-4."
            ;;
    esac
    echo ""  # Print a newline for better readability
done
```

Explanation

- **Purpose**: This script manages environment variables by allowing users to view current variables, set new ones, and display them using different commands.
- **Functions**:
  - **display_environment_variables**: Uses the `printenv` command to list all current environment variables.
  - **set_environment_variable**: Prompts the user for a variable name and value, then sets and exports the variable in the current shell session.
  - **show_env_command**: Utilizes the `env` command to show all current environment variables.
- **Menu Options**:
  - Users can select options to display environment variables, set a new variable, or exit.

Steps to Run

1. **Create the Script File**:
   - Open a terminal.
   - Create a new script file named `env_variables.sh` using a text editor:

     ```
     nano env_variables.sh
     ```

2. **Copy and Paste the Script**:
   - Paste the provided script into the editor and save the file.
3. **Make the Script Executable**:
   - Change the permission to make the script executable:

     ```
     chmod +x env_variables.sh
     ```

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

     ```
     ./env_variables.sh
     ```

5. **Follow the Prompts**:
   - The script will display a menu for the user to select options.

Sample Output

Here's a sample interaction with the script:

```
$ ./env_variables.sh
Environment Variable Manager
1. Display current environment variables using printenv
2. Set and export a new environment variable
3. Display environment variables using env
4. Exit
```

```
Select an option (1-4): 1
Current Environment Variables:
USER=your_username
HOME=/home/your_username
SHELL=/bin/bash
...

Select an option (1-4): 2
Enter the name of the environment variable: MY_VAR
Enter the value of the environment variable: HelloWorld
Environment variable 'MY_VAR' set to 'HelloWorld'.

Select an option (1-4): 1
Current Environment Variables:
USER=your_username
HOME=/home/your_username
SHELL=/bin/bash
MY_VAR=HelloWorld
...

Select an option (1-4): 3
Displaying environment variables using 'env':
USER=your_username
HOME=/home/your_username
SHELL=/bin/bash
MY_VAR=HelloWorld
...

Select an option (1-4): 4
Exiting the Environment Variable Manager. Goodbye!
```

Explanation of Output

1. **Initial Menu**:
   - The script prompts the user to choose between displaying environment variables, setting a new one, or exiting.
2. **Displaying Current Environment Variables**:
   - When option 1 is selected, the `printenv` command outputs the current environment variables and their values.
3. **Setting a New Environment Variable**:
   - Upon selecting option 2, the user is prompted for a variable name and value. The script sets this variable using `export`, making it available in the current session.
4. **Displaying Variables with `env`**:
   - Option 3 uses the `env` command to list environment variables, showing that the new variable has been set.
5. **Exiting the Script**:
   - Selecting option 4 allows the user to exit the script cleanly.

This script serves as a practical tool for managing environment variables, which are essential for configuring the behavior of processes in Unix-like operating systems.

10. Piping Commands: Develop a shell script that uses pipes to connect multiple commands. For example, use grep to filter output from ls and sort the results.

Shell Script

```
#!/bin/bash

# Shell Script for Piping Commands
# Usage: ./piping_commands.sh

# Function to display the contents of the current directory
list_directory_contents() {
    echo "Listing contents of the current directory:"
    ls -l
    echo ""
}

# Function to filter and sort directory contents
filter_and_sort_contents() {
    read -p "Enter a pattern to filter files (e.g., .txt for text files): " pattern
    echo "Filtering and sorting files with pattern '$pattern'..."

    # Use ls, grep, and sort in a pipeline
    ls -l | grep "$pattern" | sort
    echo ""
}

# Main menu for user interaction
echo "Piping Commands Manager"
echo "1. List contents of the current directory"
echo "2. Filter and sort directory contents"
echo "3. Exit"

# Loop until the user chooses to exit
while true; do
    read -p "Select an option (1-3): " option

    case $option in
        1)  # List directory contents
            list_directory_contents
            ;;
        2)  # Filter and sort directory contents
            filter_and_sort_contents
            ;;
        3)  # Exit the script
            echo "Exiting the Piping Commands Manager. Goodbye!"
            exit 0
            ;;
        *)  # Invalid option
            echo "Invalid option. Please select 1-3."
            ;;
    esac
    echo ""  # Print a newline for better readability
done
```

Explanation

- **Purpose**: This script demonstrates the use of pipes in shell scripting by allowing users to list directory contents and filter/sort them based on a specified pattern.
- **Functions**:
  - **list_directory_contents**: Displays the contents of the current directory using the `ls -l` command.
  - **filter_and_sort_contents**: Prompts the user for a pattern to filter the files, then uses a combination of `ls`, `grep`, and `sort` commands piped together to display the filtered and sorted results.
- **Menu Options**:
  - Users can select options to list directory contents, filter/sort files, or exit.

Steps to Run

1. **Create the Script File**:
   - Open a terminal.
   - Create a new script file named `piping_commands.sh` using a text editor:

     ```
     nano piping_commands.sh
     ```

2. **Copy and Paste the Script**:
   - Paste the provided script into the editor and save the file.
3. **Make the Script Executable**:
   - Change the permission to make the script executable:

     ```
     chmod +x piping_commands.sh
     ```

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

     ```
     ./piping_commands.sh
     ```

5. **Follow the Prompts**:
   - The script will display a menu for the user to select options.

Sample Output

Here's a sample interaction with the script:

```
$ ./piping_commands.sh
Piping Commands Manager
1. List contents of the current directory
2. Filter and sort directory contents
3. Exit
Select an option (1-3): 1
Listing contents of the current directory:
total 12
-rw-r--r-- 1 user user  234 Nov  1 10:00 file1.txt
-rw-r--r-- 1 user user  567 Nov  1 10:01 file2.log
-rw-r--r-- 1 user user  890 Nov  1 10:02 file3.txt

Select an option (1-3): 2
Enter a pattern to filter files (e.g., .txt for text files): .txt
Filtering and sorting files with pattern '.txt'...
```

```
-rw-r--r-- 1 user user  234 Nov  1 10:00 file1.txt
-rw-r--r-- 1 user user  890 Nov  1 10:02 file3.txt

Select an option (1-3): 3
Exiting the Piping Commands Manager. Goodbye!
```

Explanation of Output

1. **Initial Menu**:
   - The script presents options to list directory contents, filter and sort files, or exit.
2. **Listing Directory Contents**:
   - When option 1 is selected, the script runs `ls -l`, displaying the detailed contents of the current directory.
3. **Filtering and Sorting Files**:
   - Upon selecting option 2, the user is prompted to enter a pattern (e.g., `.txt`). The script then uses the pipeline:
     - `ls -l` lists the directory contents.
     - `grep "$pattern"` filters the results based on the user-provided pattern.
     - `sort` sorts the filtered results alphabetically.
4. **Exiting the Script**:
   - Choosing option 3 allows the user to exit the script cleanly.

This lab effectively illustrates how to utilize pipes in shell scripts to manipulate command outputs dynamically.

11. Handling File Descriptors: Create a shell script that redirects standard output and error to a file. Use commands like command > output.txt 2> error.txt to demonstrate file descriptor manipulation.

Shell Script

```
#!/bin/bash

# Shell Script for Handling File Descriptors
# Usage: ./file_descriptors.sh

# Define output and error files
OUTPUT_FILE="output.txt"
ERROR_FILE="error.txt"

# Function to create files and demonstrate redirection
redirect_output() {
    echo "Demonstrating standard output and error redirection."

    # Clear previous output files
    > "$OUTPUT_FILE"
    > "$ERROR_FILE"

    # Generate output and error
    echo "This is a standard output message." > "$OUTPUT_FILE"
    echo "This is a standard error message." 1>&2 > "$ERROR_FILE"

    # Simulating a command that generates an error
    echo "Trying to list a non-existent directory:" >> "$OUTPUT_FILE"
    ls /non_existent_directory >> "$OUTPUT_FILE" 2>> "$ERROR_FILE"
}

# Function to display the contents of output and error files
display_results() {
    echo ""
    echo "Contents of $OUTPUT_FILE:"
    cat "$OUTPUT_FILE"
    echo ""

    echo "Contents of $ERROR_FILE:"
    cat "$ERROR_FILE"
    echo ""
}

# Main execution
redirect_output
display_results
```

Explanation

- **Purpose**: This script demonstrates how to redirect standard output and standard error to separate files using file descriptors.
- **File Descriptors**:
  - `1` refers to standard output (stdout).
  - `2` refers to standard error (stderr).
- **Functions**:
  - **redirect_output**:

        - Clears any previous contents of the output and error files.
        - Writes a standard output message to `output.txt`.
        - Writes a standard error message to `error.txt`.
        - Simulates an error by attempting to list a non-existent directory, redirecting the error to `error.txt`.
    - **display_results**:
        - Displays the contents of both `output.txt` and `error.txt`.

Steps to Run

1. **Create the Script File**:
    - Open a terminal.
    - Create a new script file named `file_descriptors.sh` using a text editor:

      ```
      nano file_descriptors.sh
      ```

2. **Copy and Paste the Script**:
    - Paste the provided script into the editor and save the file.
3. **Make the Script Executable**:
    - Change the permission to make the script executable:

      ```
      chmod +x file_descriptors.sh
      ```

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

      ```
      ./file_descriptors.sh
      ```

Sample Output

Here's a sample interaction with the script:

```
$ ./file_descriptors.sh

Demonstrating standard output and error redirection.

Contents of output.txt:
This is a standard output message.
Trying to list a non-existent directory:

Contents of error.txt:
ls: cannot access '/non_existent_directory': No such file or directory
```

Explanation of Output

1. **Redirection Messages**:
    - The script begins by stating that it is demonstrating standard output and error redirection.
2. **Contents of `output.txt`**:
    - The first line contains a standard output message: "This is a standard output message."
    - The second line states that the script attempted to list a non-existent directory, which indicates where the standard output stream was redirected to.
3. **Contents of `error.txt`**:

- The error message generated by the `ls` command when trying to access a non-existent directory is captured in this file, demonstrating how standard error is handled separately from standard output.

This lab effectively illustrates file descriptor manipulation in shell scripting, showing how to separate output and error streams for better error handling and logging.

12. Signal Handling: Write a shell script that traps signals (like SIGINT) and executes a custom handler. Use the trap command to manage signals and demonstrate their effect on process termination.

Shell Script

```
#!/bin/bash

# Shell Script for Advanced Signal Handling
# Usage: ./advanced_signal_handling.sh

# Log file to store signal handling messages
LOG_FILE="signal_handling.log"

# Custom signal handler function for SIGINT (Ctrl+C)
handle_sigint() {
    echo "SIGINT (Ctrl+C) received! Cleaning up..."
    echo "$(date): SIGINT received, exiting gracefully." >> "$LOG_FILE"
    exit 0
}

# Custom signal handler function for SIGTERM
handle_sigterm() {
    echo "SIGTERM signal received! Terminating..."
    echo "$(date): SIGTERM received, exiting gracefully." >> "$LOG_FILE"
    exit 0
}

# Custom signal handler function for SIGHUP
handle_sighup() {
    echo "SIGHUP signal received! Reloading configuration..."
    echo "$(date): SIGHUP received, reloading configuration." >> "$LOG_FILE"
}

# Trap multiple signals and link them to their respective handlers
trap 'handle_sigint' SIGINT
trap 'handle_sigterm' SIGTERM
trap 'handle_sighup' SIGHUP

# Main loop
echo "Running... Press Ctrl+C to trigger SIGINT, or send SIGTERM (kill) to
terminate."

# Infinite loop to keep the script running
while true; do
    # Simulating some ongoing process
    echo "Working... (Press Ctrl+C to stop, or send SIGTERM to terminate)"
    sleep 2
done
```

Explanation

- **Purpose**: This script demonstrates advanced signal handling, including logging of received signals to a log file and providing handlers for multiple types of signals.
- **Signal Handlers**:
    - **SIGINT**: Triggered by Ctrl+C; the handler cleans up and exits gracefully.
    - **SIGTERM**: Allows for external termination of the script using the `kill` command.

    - **SIGHUP**: Typically sent when a terminal session is closed; it can be used to reload configurations instead of terminating the script.
- **Logging**:
    - Each handler logs the date and signal received to a log file named `signal_handling.log`, providing a record of events.
- **trap Command**:
    - Multiple signals are trapped and linked to their respective handlers, allowing for differentiated handling based on the signal received.
- **Main Loop**:
    - The script enters an infinite loop, simulating ongoing work with a sleep interval. It displays messages indicating the current status and available actions.

Steps to Run

1. **Create the Script File**:
    - Open a terminal.
    - Create a new script file named `advanced_signal_handling.sh` using a text editor:

      ```
      nano advanced_signal_handling.sh
      ```

2. **Copy and Paste the Script**:
    - Paste the provided script into the editor and save the file.
3. **Make the Script Executable**:
    - Change the permission to make the script executable:

      ```
      chmod +x advanced_signal_handling.sh
      ```

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

      ```
      ./advanced_signal_handling.sh
      ```

5. **Testing the Signals**:
    - **Ctrl+C**: Press Ctrl+C to trigger the SIGINT signal.
    - **SIGTERM**: In a new terminal, find the PID of the script using `ps` and then send the SIGTERM signal:

      ```
      kill -TERM <PID>
      ```

Sample Output

Here's a sample interaction with the script:

```
$ ./advanced_signal_handling.sh
Running... Press Ctrl+C to trigger SIGINT, or send SIGTERM (kill) to terminate.
Working... (Press Ctrl+C to stop, or send SIGTERM to terminate)
Working... (Press Ctrl+C to stop, or send SIGTERM to terminate)
^CSIGINT (Ctrl+C) received! Cleaning up...
```

Log File Example (`signal_handling.log`)

After pressing Ctrl+C, the log file will contain:

```
2024-11-01 12:00:00: SIGINT received, exiting gracefully.
```

If you send a SIGTERM signal instead:

```
$ kill -TERM <PID>
```

The output will show:

```
SIGTERM signal received! Terminating...
```

Explanation of Output

1. **Signal Handling Messages**:
   - The script informs the user about the current actions and signals that can be triggered.
2. **Signal Responses**:
   - When a SIGINT signal is received (by pressing Ctrl+C), the script logs the event and exits gracefully.
   - When a SIGTERM signal is sent, the corresponding handler is executed, logging the termination event.
3. **Logging**:
   - The log file records the date and type of each received signal, providing a persistent record for monitoring and debugging.

This advanced version of the script effectively demonstrates robust signal handling in shell scripting, along with graceful exits and logging for better process management.

LAB VIVA QUESTION BANK

1. What is the purpose of the fork() system call in process creation?

Sample Answer: The fork() system call is used to create a new process by duplicating the existing process. The new process, called the child process, is a copy of the parent process and has its own unique process ID.

2. How do you bring a background process to the foreground in a shell script?

Sample Answer: To bring a background process to the foreground, you can use the fg command followed by the job number, like fg %1. This resumes the specified job in the foreground.

3. What is the difference between bg and fg commands?

Sample Answer: The bg command resumes a suspended job in the background, allowing the shell to accept new commands while the job runs. The fg command, on the other hand, brings a background job to the foreground, allowing the user to interact with it directly.

4. Describe how to list current jobs in a shell session.

Sample Answer: You can list current jobs in a shell session using the jobs command. It shows the status of all jobs started in the current session, along with their job numbers.

5. How do you check memory usage on a Unix system?

Sample Answer: Memory usage can be checked using the free command, which provides an overview of total, used, and available memory. Additionally, the top command can be used to display dynamic memory usage along with other system statistics.

6. What commands are used to create, copy, move, and delete files in a shell script?

Sample Answer: The commands used are touch to create files, cp to copy files, mv to move or rename files, and rm to delete files.

7. How can you modify file permissions and ownership in Unix?

Sample Answer: File permissions can be modified using the chmod command, while ownership can be changed using the chown command. These commands allow users to specify read, write, and execute permissions for user, group, and others.

8. Explain the purpose of the sleep command in job scheduling.

Sample Answer: The sleep command is used to introduce a delay in the execution of a script. This is useful in job scheduling to pause between command executions, allowing for better control over the timing of tasks.

9. How do environment variables affect shell scripts?

Sample Answer: Environment variables store configuration values that can influence the behavior of scripts and applications. They provide a way to pass configuration data and control the environment in which scripts run.

10. What is piping in Unix, and how is it used in shell scripts?

Sample Answer: Piping is a technique used to pass the output of one command as input to another command using the | operator. It allows for chaining commands together to perform complex operations more efficiently.

11. Describe how to redirect standard output and error in a shell script.

Sample Answer: Standard output can be redirected using > to write to a file, while standard error can be redirected using 2>. For example, command > output.txt 2> error.txt redirects the output to output.txt and errors to error.txt.

12. How can signals be trapped in a shell script?

Sample Answer: Signals can be trapped using the trap command followed by a handler function and the signal name. This allows the script to execute specific actions when certain signals are received, like cleaning up resources before exiting.

13. What happens when you send a SIGINT signal to a running process?

Sample Answer: When a SIGINT signal (triggered by Ctrl+C) is sent to a running process, it typically interrupts the process, causing it to terminate unless a specific signal handler is defined to manage it gracefully.

14. What command would you use to display all environment variables in the current shell?

Sample Answer: The command printenv or env can be used to display all environment variables in the current shell, showing their names and values.

15. How do you create a new directory in a shell script?

Sample Answer: A new directory can be created using the mkdir command followed by the desired directory name. For example, mkdir new_directory creates a directory named new_directory.

16. What is the significance of using the -p option with the mkdir command?

Sample Answer: The -p option allows the creation of parent directories as needed. If the specified directory hierarchy does not exist, it will create all necessary parent directories without throwing an error.

17. How can you check if a file exists in a shell script?

Sample Answer: You can check if a file exists using the -e flag in a conditional statement. For example:

```
if [ -e "filename.txt" ]; then
```

```
    echo "File exists."
else
    echo "File does not exist."
fi
```

18. Explain how to use the `cp` command with the `-r` option.

Sample Answer: The `-r` option with the `cp` command allows for recursive copying of directories and their contents. For example, `cp -r source_directory target_directory` copies the entire `source_directory` and all its contents to `target_directory`.

19. What is the function of the `ls` command in file operations?

Sample Answer: The `ls` command is used to list the contents of a directory. It can display files and subdirectories along with their attributes, such as permissions, ownership, size, and modification date.

20. How can you safely delete a file or directory in a script?

Sample Answer: To safely delete a file or directory, you can use the `rm` command for files and `rmdir` for empty directories. It is often prudent to prompt the user for confirmation before deletion to prevent accidental data loss.

21. What is the role of the `top` command in monitoring system resources?

Sample Answer: The `top` command displays real-time information about system processes, CPU usage, memory usage, and other system statistics. It allows users to monitor system performance and resource utilization dynamically.

22. How do you change the ownership of a file to a specific user and group?

Sample Answer: You can change the ownership of a file using the `chown` command followed by the user and group names and the file name. For example, `chown user:group filename.txt` changes the ownership to `user` and group `group`.

23. What does the `kill` command do, and how is it used?

Sample Answer: The `kill` command is used to send signals to processes, typically to terminate them. It can be invoked using the process ID (PID) to specify which process to send a signal to, such as `kill <PID>` for a default SIGTERM signal.

24. Describe how to handle errors gracefully in a shell script.

Sample Answer: Error handling can be done using conditional checks to verify the success of commands. Using constructs like `if`, `||`, or `trap` allows the script to respond appropriately to errors, such as displaying messages or cleaning up resources.

25. How can you check the exit status of the last executed command in a shell script?

Sample Answer: The exit status of the last executed command can be checked using the special variable `$?`. A value of `0` indicates success, while any non-zero value indicates an error.

26. What is the difference between hard links and symbolic links in Unix?

Sample Answer: Hard links point directly to the inode of a file, making them indistinguishable from the original file. Symbolic links, or symlinks, point to the file name and can link to files on different file systems but can break if the original file is deleted.

27. How do you execute multiple commands in a single line in a shell?

Sample Answer: Multiple commands can be executed in a single line by separating them with semicolons ;. For example, `command1; command2; command3` executes each command sequentially.

28. What are some common signals used in Unix, and what do they represent?

Sample Answer: Common signals include:

- SIGINT (2): Interrupt signal (Ctrl+C).
- SIGTERM (15): Termination signal.
- SIGHUP (1): Hangup signal, typically sent when a terminal session ends.
- SIGKILL (9): Forceful termination of a process that cannot be caught or ignored.

29. How do you set an environment variable in a shell script?

Sample Answer: An environment variable can be set using the syntax `VARIABLE_NAME=value`. For example, `MY_VAR="Hello World"` sets an environment variable named `MY_VAR` with the value "Hello World".

30. What is the significance of using && and || in shell scripts?

Sample Answer: The && operator allows you to execute a command only if the previous command succeeded (exit status 0). Conversely, the || operator executes a command only if the previous command failed (non-zero exit status). This allows for conditional execution based on success or failure.