CL2006 - Operating Systems Spring 2024
LAB # 3 MANUAL (Common)

Please note that all labs' topics including pre-lab, in-lab and post-lab exercises are part of the theory and labsyllabus. These topics will be part of your Midterms and Final Exams of lab and theory.

Objectives:

1. Use your existing knowledge of Linux command line and C programming (PF, OOPs, Data Structures) to learn linux bash shell scripting to automate basic Linux jobs.

2. You should be able to write an error free Linux bash shell script given a problem description.

Lab Tasks:

1. Task # 1: Automatic file backup
2. Task # 2: Automatic data processing

Try https://www.onlinegdb.com/online_bash_shell if you are doing the task online.

Delivery of Lab contents:

Strictly following the following content delivery strategy. Ask students to take notes during the lab.

1st Hour

- Pre-Lab (up to 15 minutes)
- Explain importance and history of shell scripting. Command-line vs Shell Scripting. (15 minutes)
- Ask students to type of run Task # 1 (15 minutes). Observe their weaknesses and back scripting command which you can cover from Handout # 1 and/or Handout # 2.

2nd Hour

- Cover Handout # 2. Few of you can cover details from Handout # 1 (keep this session within 45 minutes).
- Ask students to type and execute Task # 2 (15 minutes)

3rd Hour

- Devote full hour doing the in-lab problem. (60 minutes)

Created by: Nadeem Kafi (01/02/2024)
DEPARTMENT OF COMPUTER SCEICEN, FAST-NU, KARACHI

** ChatGPT is heavily used to make the contents of this document along with other Internet sources.

EXPERIMENT 3
Creating, Executing LINUX Bash Shell Script

With the widespread adoption of Linux, shell scripting became an essential tool for automating tasks, executing multiple commands sequentially, and performing various system administration tasks. Bash scripting became the most used scripting language on Linux systems due to its availability and compatibility with POSIX standards. A Linux Bash shell script is a text file containing a series of commands written in the Bash scripting language. Bash (Bourne Again Shell) is a popular command-line interpreter and scripting language for Unix-like operating systems, including Linux. Bash scripts can incorporate control structures such as loops and conditional statements, variables, functions, and command-line arguments to enhance their functionality.

History of different shells used in Unix and Linux

The history of Linux shell scripting is closely intertwined with the history of Unix, upon which Linux is based. Shell scripting on Unix-like systems dates back to the early days of Unix development in the 1970s. Here's a brief overview:

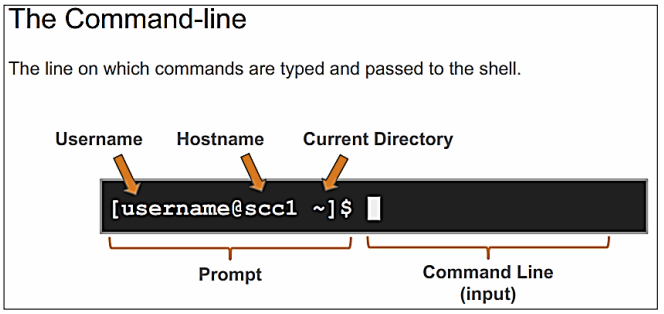
1. Bourne Shell (sh):
   - Developed by Stephen Bourne at AT&T Bell Laboratories in the early 1970s.
   - Became the default shell for Unix systems.
   - Provided basic scripting capabilities with loops, conditionals, and command execution.

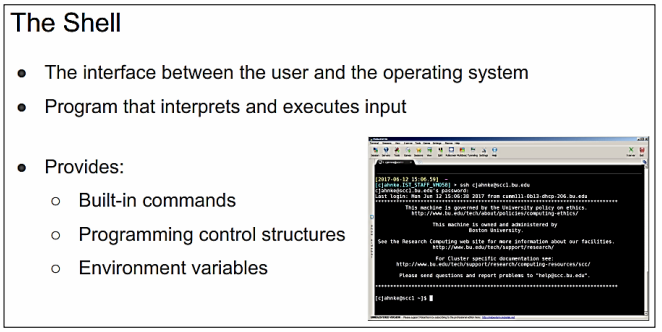
2. C Shell (csh):
   - Developed by Bill Joy at the University of California, Berkeley, in the late 1970s.
   - Featured a C-like syntax and interactive features such as command history and job control.
   - Popular among users who preferred its interactive features.

3. Korn Shell (ksh):
   - Developed by David Korn at AT&T Bell Laboratories in the early 1980s.
   - Combined features of both Bourne Shell and C Shell.
   - Introduced advanced scripting features such as associative arrays, arithmetic evaluation, and built-in string manipulation.

4. Bash (Bourne Again Shell):
   - Developed by Brian Fox for the GNU Project in 1989.
   - Based on the Bourne Shell.
   - Became the default shell for many Linux distributions.
   - Added features such as command-line editing, history, job control, and enhanced scripting capabilities.
   - Became the de facto standard shell for Linux systems due to its ubiquity and powerful scripting capabilities.

5. Other Shells:
   - Several other shells exist, including tcsh (an enhanced version of C Shell), zsh (Z Shell), and fish (Friendly Interactive Shell), each with its own features and syntax.





Learn Linux Bash shell scripts by real-world examples

Script # 1: Automating File Backup

This script is designed to copy all files from a source directory (`$SRC_DIR`) to a destination directory (`$DST_DIR`). It first checks if the destination directory exists. If it does not, it creates it. Then, it iterates over each file in the source directory and copies it to the destination directory. If the source directory does not exist, it prints an error message and exits with a non-zero status code.

```
#!/bin/bash

SRC_DIR=/path/to/source/directory
DST_DIR=/path/to/backup/directory

if [ ! -d "$DST_DIR" ]; then
   mkdir -p "$DST_DIR"
fi

for file in "$SRC_DIR"/*; do
   cp "$file" "$DST_DIR"
done

if [ ! -d "$SRC_DIR" ]; then
   echo "Error: Source directory does not exist"
   exit 1
fi
```

Description

1. Variable Declarations:
   - `SRC_DIR=/path/to/source/directory`: Defines a variable `SRC_DIR` containing the path to the source directory.
   - `DST_DIR=/path/to/backup/directory`: Defines a variable `DST_DIR` containing the path to the backup directory.

2. Directory Existence Check and Creation:
   - `if [ ! -d "$DST_DIR" ]; then`: Checks if the destination directory (`$DST_DIR`) does not exist.
   - `mkdir -p "$DST_DIR"`: If the destination directory does not exist, creates it (including any necessary parent directories) using the `mkdir` command with the `-p` option.

3. File Copying using a Loop:
   - `for file in "$SRC_DIR"/*; do`: Iterates over each file in the source directory (`$SRC_DIR`) using a `for` loop.
   - `cp "$file" "$DST_DIR"`: Copies each file from the source directory to the destination directory (`$DST_DIR`) using the `cp` command.

4. Error Handling:
   - `if [ ! -d "$SRC_DIR" ]; then`: Checks if the source directory (`$SRC_DIR`) does not exist.
   - `echo "Error: Source directory does not exist"`: If the source directory does not exist, prints an error message.
   - `exit 1`: Exits the script with a non-zero status code (1) to indicate an error.

Script # 2: Automating Data Processing

This script defines a function process_data that takes an input file as an argument, processes the data in the file using command-line tools (cut, grep, sort), and returns the processed data. It then loops through each .txt file in the specified directory, calls the process_data function on each file, captures the processed data, and writes it to a new file with the suffix _processed.txt. This script demonstrates how to define and call functions in Bash, pass arguments to functions, process data using command-line tools within a function, and use loops to iterate over files in a directory.

```
#!/bin/bash
process_data() {
    input_file=$1
    output_file=$(cut -f 1,3 $input_file | grep 'foo' | sort -n)
    echo $output_file
}
for file in /path/to/files/*.txt; do
    processed_data=$(process_data $file)
    echo $processed_data > "${file}_processed.txt"
done
```

Description

1. Function Definition (`process_data`):
   - `process_data() { ... }`: Defines a function named `process_data` for processing input data.

2. Function Argument:
   - `input_file=$1`: Assigns the first argument passed to the function (`$1`) to the variable `input_file`.

3. Data Processing:
   - `output_file=$(cut -f 1,3 $input_file | grep 'foo' | sort -n)`: Processes the data from the input file using a series of command-line tools:
   - `cut -f 1,3 $input_file`: Extracts the first and third fields from the input file using the `cut` command.
   - `grep 'foo'`: Filters the extracted data to include only lines containing the string 'foo' using the `grep` command.
   - `sort -n`: Sorts the filtered data numerically using the `sort` command.

4. Returning Processed Data:
   - `echo $output_file`: Prints the processed data to standard output, which will be captured when the function is called.

5. Function Invocation in a Loop:
   - `for file in /path/to/files/*.txt; do ... done`: Iterates over each `.txt` file in the specified directory.

6. Function Call:
   - `processed_data=$(process_data $file)`: Calls the `process_data` function with the current file as an argument (`$file`) and captures the output in the variable `processed_data`.

7. Writing Processed Data to a New File:
   - `echo $processed_data > "${file}_processed.txt"`: Writes the processed data to a new file named `${file}_processed.txt`, where `${file}` is the name of the original file being processed.

In-Lab

Consider the following log file data entries:

192.168.1.1 - - [01/Jan/2024:12:00:00 +0000] "GET /page1 HTTP/1.1" 200 1234
192.168.1.2 - - [01/Jan/2024:12:01:00 +0000] "GET /page2 HTTP/1.1" 404 5678
192.168.1.3 - - [01/Jan/2024:12:02:00 +0000] "GET /page1 HTTP/1.1" 200 9876
192.168.1.1 - - [01/Jan/2024:12:03:00 +0000] "POST /page3 HTTP/1.1" 301 2345
192.168.1.2 - - [01/Jan/2024:12:04:00 +0000] "GET /page2 HTTP/1.1" 200 8765
192.168.1.3 - - [01/Jan/2024:12:05:00 +0000] "GET /page4 HTTP/1.1" 404 3456
192.168.1.1 - - [01/Jan/2024:12:06:00 +0000] "GET /page1 HTTP/1.1" 200 6789
192.168.1.2 - - [01/Jan/2024:12:07:00 +0000] "GET /page2 HTTP/1.1" 200 1234

Now you are tasked with creating a shell script to analyze log files generated by a web server. The script should perform the following tasks:

- Accept a log file as input.
- Count the total number of requests (lines) in the log file.
- Determine the number of unique IP addresses that made requests.
- Identify the top 5 most frequent IP addresses and their corresponding request counts.
- Calculate the total size of data transferred (in bytes) from the log file.
- Generate a summary report containing the above information and save it to an output file.

Sample solution:

```
#!/bin/bash
# Log File Analyzer
# Accept log file as input
log_file="$1"
# Count total number of requests
total_requests=$(wc -l < "$log_file")
# Determine number of unique IP addresses
unique_ips=$(awk '{print $1}' "$log_file" | sort -u | wc -l)
# Identify top 5 most frequent IP addresses
top_ips=$(awk '{print $1}' "$log_file" | sort | uniq -c | sort -nr | head -5)
# Calculate total size of data transferred
total_size=$(awk '{sum += $10} END {print sum}' "$log_file")
# Generate summary report
echo "Log File Analysis Report" > analysis_report.txt
echo "------------------------" >> analysis_report.txt
echo "Total Requests: $total_requests" >> analysis_report.txt
echo "Unique IP Addresses: $unique_ips" >> analysis_report.txt
echo "Top 5 IP Addresses:" >> analysis_report.txt
echo "$top_ips" >> analysis_report.txt
echo "Total Size of Data Transferred: $total_size bytes" >> analysis_report.txt
echo "Report generated on: $(date)" >> analysis_report.txt
```

Post-Lab

1. Problem Statement 1: File Renaming Utility

You are tasked with creating a bash script that renames multiple files in a directory according to a specified naming convention. The script should:

a. Accept two arguments: the directory path containing the files and the new file name pattern.
b. Rename each file in the directory by appending a sequential number to the new file name pattern (e.g., `file1.txt`, `file2.txt`, etc.).
c. Preserve the original file extension during the renaming process.
d. Provide feedback to the user about the renaming process, including any errors encountered.

2. Problem Statement 2: Directory Cleanup Script

Develop a bash script to automate directory cleanup tasks by removing old files and directories. The script should:

a. Accept a directory path as an argument.
b. Identify and delete files older than a specified number of days.
c. Recursively remove empty directories within the specified directory.
d. Provide feedback to the user about the cleanup process, including the number of files and directories removed.

3. Problem Statement 3: System Monitoring Script

Create a bash script to monitor system resources and generate a report. The script should:

a. Collect information about CPU usage, memory usage, disk space, and network traffic.
b. Calculate average values for each resource over a specified time period.
c. Generate a report containing the collected data and average values.
d. Provide options for the user to customize the time period and output format of the report.