Assignment
Course: IN616 – Operating Systems Concepts

Semester 1, 2021

Otago Polytechnic

Assignment Due Date: Monday, 31 **May 2021, 11:59 pm**
Weighting: 30% of overall marks
Learning outcomes covered: ALL

1 Instructions

In this assignment you will write a collection of bash scripts that help automate general administrative tasks on an Ubuntu Linux server. The assignment is comprised of the following components:

- **Task 1:** Author a script to automate the creation of users and configuration of the user environment **(60%)**
- **Task 2:** Author a script to backup directories and upload the backup to a remote server **(20%)**
- **Formal Aspects:** Error handling, documentation, commenting, code modularity and git usage **(20%)**

2 Assignment Virtual Machine

You must perform development of your scripts in a specifically provided virtual machine available on vRealize. You have been provided a blueprint on vRealize named `IN617-BSA` which you are to request. A dedicated virtual machine has been provided so that you keep you Bash Scripting Assignment separate from your other virtual machines. Also, the `IN617-BSA-2019` virtual machine has better security and enforces a custom password for each user. The credentials for the virtual machine are:

- **Username:** `student`
- **Password:** `iambatman2001`

NOTE: You will be required to change the password at the first login.

The `IN617-BSA-2019` virtual machine has SSH installed by default, so you can connect using PuTTY from within the OP campus. You can connect off-campus using PuTTY by pivoting through the Kate web server. PuTTY connection via SSH is beneficial as it provides the ability to copy/paste.

The `IN617-BSA-2019` virtual machine is also accessible using the vRealize web interface and connection via VMRC or Remote Console. Note: You must have VMRC installed (already available on the OP lab computers). Note: You must have the OP SSL certificate installed to use Remote Console (already available using Google Chrome in the OP lab computers). Note: You will not be able to copy/paste using Remote Console or VMRC.

3 Task 1 – Creating a User Environment (60%)

Task 1 in this assignment specifies that you will write a script that automates the process of user creation, followed by configuration of the user environment on a Ubuntu Linux system. The script will ingest a file containing user-related information, and should create users based on the information in the provided file. Additionally, the script should perform a basic configuration of the user environment including creation of secondary groups, shared folders and setting of permissions.

3.1 Core Functionality

The core functionality required in your Bash script is listed below:

- The script should ingest a CSV file (local or remote) that contains user information, which contains the values required to perform the following tasks:
- Create each specified user while implementing best practices
- Set a default password for the user based on the user's birth date
- Create the required secondary groups (if specified and non-existent)
- Assignment to the correct secondary groups (if specified)
- Create the required shared folder (if specified and non-existent)
- Create the required secondary group for the shared folder (if specified and non-existent)
- Assignment of the user to a group to access the shared folder (if specified)
- Creation of a link to the users shared folder (if specified)
- Creation of an alias to shutdown the system (if user is in the `sudo` group)
- Enforce a password change at first login

3.1.1 Script Input File

As stated, your script should ingest a CSV file containing user data. Your script should be able to handle the following inputs:

1. Support for two different command line arguments, either:
   (a) A URI for a web-based resource
   (b) A local file system location
2. Support for two different user inputs when no command line argument have been supplied, either:
   (a) A URI for a web-based resource
   (b) A local file system location

When a file has been provided in the local file system, it should be checked for existence and that it is actually a parse-able file. If a file is provided by a URI, the script should download the file (to the same directory as the script), check file existence, that it is parse-able and then proceed with processing.

Once the input file is ingested, the script should parse the file and use it to create users and configure the environment on the local system. One sample file has been provided as an example of the structure of the input file format. The example files are available from:

http://kate.ict.op.ac.nz/~faisalh/IN617linux/users.csv

The contents of the example fil are listed below:

```
e-mail;birth date;groups;sharedFolder
edsger.dijkstra@tue.nl;1930/05/11;sudo,staff;/staffData
john.mccarthy@caltech.edu;1927/09/04;sudo,visitor;/visitorData
andrew.tanenbaum@vua.nl;1944/03/16;staff;/staffData
alan.turing@cam.ac.uk;1912/06/23;visitor;/visitorData
linus.torvalds@linux.org;1969/12/28;sudo;
bjarne.stroustroup@tamu.edu;1950/12/30;;/visitorData
ken.thompson@google.com;1973/02/04;sudo,visitor;
james.gosling@sun.com;1955/05/19;staff;/staffData
tim.berners-lee@mit.edu;1955/06/08;sudo,visitor;/visitorData
```

Carefully check the format of the provided file and use it as an example of what parameters your script must be able to handle. Each row of the sample file represents an individual user. The sample file contains a header that documents each column in the file. The available columns in the file are:

- The email address of the user
- The birth date of the user (in the format YYYY/MM/DD, for example, 1991/11/17)
- The secondary groups the user should be added to
- A shared folder that the user requires full access to (full rwx permissions)

3.1.2 User Environment Requirements

Username: The *username* is to be generated from the provided e-mail address. The username structure must adhere to the following conventions: The first letter of the first name, followed by the entire surname. For example, if a user had the following email: `linus.torvalds@linux.org`, the username would be: `ltorvalds`. If a given username already exists, the script should ignore the user entry in the file (i.e. not create the user) and provide console notification about this event.

Password: The *password* is to be generated from the birth date. The default password for the user should be the month and year of the users birth date appended together in the following format: `MMYYYY`. For example, if a user Linus Torvalds has the birth date 1969/12/28, his default password should be `121969`. Check the `date` command on how extract date components. The user should also be asked to change his/her password upon first login. For this purpose have a look at the `chage` command.

Shared Folder: The *shared folder* should be created (if not yet existing) and permissions adjusted accordingly. The shared folder paths should be treated as absolute paths. To maintain the permission set for existing users to the shared folder (not just the last added user), you will need to think about how to use additional groups (beyond the ones specified in the users file) to achieve this. The owning user of the shared folder should either be root or the user who is running the script. Note: users in the *other* group should have no access to shared folders.

Share Folder Link: For each user with any permission to a shared directory, place a link within the user's home folder to the shared directory. The link should be named `shared` and point to the shared folder the user has access to.

Alias: For each user with `sudo` access, create an *alias* named `off` that shuts down the system (using the command: `systemctl poweroff`). The alias should be stored in the correct file so that it is loaded upon every login to ensure the alias is available to the user.

Best Practice: It is essential that you implement best practices for user, group, shared folder, link and alias creation and configuration. We have discussed this in class. For example, some best practice for user creation are home directory naming convention and default shell settings. For example, the best practice for alias creation are using the `.bash_alias` file for storing aliases.

3.1.3 Script Output and Logging

Script Output: Your script should provide console output (stdout) for **major events**. This is to provide the user with information about what actions the script is performing. The following list provides a summary of the required output:

- Print information about the input file
- Print a dialog about the number of users to be added
- For each user, print a summary of the created user environment including (but not limited to): username, home directory, shared folder, shared folder link, alias
- Print information when a major error is encountered, including (but not limited to): user creation failed, shared folder creation failed, group creation failed, alias creation failed, link creation failed

Before the script starts creating users and setting up the user environment, it should present the user a dialogue that informs about the number of to-be-added users and **asks for confirmation before proceeding**. Try not to use too much console-based output, if it is too cluttered the user will not be able to easily track what is happening. Also, formatting console-based output using line breaks or indentation can greatly increase readability.

Logging Output: Your script should also provide functionality to output additional detailed information to a log file. This log file should be much more detailed than the console output, and should keep track of the success/failure of user creation and success/failure of user environment configuration. This is specified so that the user can review the log file to determine where anything went wrong when running the script. This log file should include date information that indicates when the script was run. Make sure you also choose a sensible name for the log file.

3.2 Script Error Checking

Your script should check for all possible errors and deal with them by presenting the user with information and also logging it. Carefully think about things that can possibly go wrong or cause problems during script execution. Acknowledge possible problems in the script, even if you do not address them – that way your tutor knows you have identified the error, but have not fixed it.

3.3 Code Structure and Best Practice

Code Structure: Structure your code by moving repetitive code into functions. When implementing functions, use local variables where possible to avoid eventual side effects of global variable usage.

Code Comments: Provide sensible documentation through comments. *Sensible* implies that you don't necessarily comment each line of code, but comment critical or complex operations that might be difficult for another user to understand quickly when reviewing your code. This way the reader should get quick insight about what happens where in your code.

3.4 Hints

The following hints may provide useful during development of your script:

- Start with a basic script that reads the input URI and local file
- Start by using print statements instead of actual commands
- Create a *dummy user input file* that only contains one user for testing
- Test added code thoroughly before adding any additional functionality
- Create a *clean-up* script that removes users, groups and shared folders so that you can quickly reset your system back to default to ease the testing process
- If you don't manage to get all parts of the code working, annotate sections that are missing code or functionality and describe what should have happened there
- Continually save code changes using git (discussed later)

4 Task 2 – Backup Script (20%)

For task 2 in this assignment you will write a script that compresses a given folder and uploads it to a remote location – this is essentially a backup script. You should be able to use this script to backup the contents of any directory in the local file system.

4.1 Core Functionality

The core features of the script are:

- The script should ingest a directory name as an argument, or ask the user for a directory if no argument is provided
- The script should generate a compressed tarball archive in gunzip format (`.tar.gz` file) that contains the entire contents of the supplied directory
- The script should prompt the user during execution for the following details:
  - IP address or URL of the remote server
  - Port number of the remote server
  - Target directory to save the compressed tarball archive
- The script should then upload the compressed tarball archive to a remote host using the **scp** protocol and details provided by the user input

As stated, your script should ingest a file system directory. Your script should be able to handle the following inputs:

1. A directory passed as a command line argument
2. A directory passed as user input when no command line argument has been supplied

The script should check the user input is a valid and that the directory exists. Then the script should generate a compressed tarball archive (with a file extension of `.tar.gz`) of the target directory and all the contents of the directory. The basename of the compressed tarball archive should be the same as the name of the input directory. Hint: Check the `tar` command for this purpose, and we will discuss this in class as well.

The script should then upload the generated archive to a remote host using secure copy, achieved using the `scp` command. The details of the remote server should be collected from the user by reading input during the script execution (use the `read` command). The script should test whether the remote server exists and if the file transfer was successful.

4.2 Error Checking

Similar to the script for Task 1, try to identify and capture all possible errors during execution (e.g. network outage, incorrect password, unknown upload directory etc.). Also be sure to notify the user and interrupt execution when any error is encountered.

4.3 Hints

The following hints may provide useful during development of your script:

- Test your script by using your *TrainingVM*, this machine has the required software package to use secure copy
- Try creating compressed tarball archives in gunzip format from the command line before trying to script the process
- Try using `scp` from the command line before trying to script the process
- Make sure to check the tarball contents to ensure the directory is being correctly archived

5 Formal Aspects – Documentation and Git (20%)

This assignment includes a collection of formal aspects including requirements for project structure and project documentation. In addition, you are required to use git for source code revision control and assignment submission.

5.1 Source Code Revision Control and Git Usage

This assessment prescribes the use of git for source code revision control, as well as for assessment submission. You have been supplied a dedicated private GitLab repository for this assignment. Your GitLab repository is hosted on the BIT GitLab website and available at: `https://gitlab.op-bit.nz/`. Specifically, you have been supplied with a repository named `IN617_S2` which will be used for storage and submission of you project source code and documentation. Your repository is available from:

`https://gitlab.op-bit.nz/BIT/2019/IN617_S2/<username>`

In the above URL, you can replace the `<username>` placeholder with your OP student login name – formatted in capital letters. You are required to use this git repository and adhere to git best practices. This includes:

- Commiting your code at logical and regular intervals
- Use of descriptive commit messages

5.2 Project Structure

Your git repository content and project source code should follow a uniform structure. You are required to adhere to the following project structure:

```
│ README.md
├─BSA_Self_Assessment.txt
├─task1
│  └─(files related to task1)
└─task2
   └─(files related to task2)
```

5.3 Project Documentation

You must include documentation to accompany your assignment. Therefore, write a brief `README` file that has the following requirements:

- Must be written in Markdown syntax
- Must be named `README.md`
- Must be located in the base directory in your git repository

Your `README.md` file must contain:

- Author details:
  - Your full name
  - You student code (aka. login name)
  - A timestamp of last date when changes were made
- Project details for each of the two scripts:
  - Summary of the purpose of the script
  - Pre-requisites of running the script
  - Instructions on how to run the script, including example commands

When writing your README file, think about the reader as someone who does not know the instructions provided in this document. Make sure to write the documentation for someone who has limited knowledge of the Linux operating system. Additionally, think of yourself half a year down the road, and what documentation you will need to understand the scripts you have authored, how to run them, and what their purposes are.

NOTE: The other components of the formal aspects of this assignment (including error handling, code commenting and code modularity) are all assessed during review of your scripts for Task 1 and Task 2.

5.4 Assignment Submission

Before submission carefully check that all your scripts work properly. It is very important to perform thorough testing of your scripts to ensure functionality and error checking capabilities.

You assignment submission is via GitLab. Technically, you do not need to submit your assignment, as the tutor will download (clone) your GitLab repository at the time and date when the assignment is due. Make sure you GitLab repository is kept up-to-date and is ready at the submission deadline.

In addition, include this document with the submission. Make sure you also complete the self-assessment worksheet (`BSA_Self_Assessment.txt`) which will be provided for download. You can write any comments you wish the marker to know. Include the document in the root folder of the submitted archive (as specified in the project structure).

	Activity		**Points**	**Completed**[*]	**Result**
Task 1	***Reading local file / URL***		5		
	Self-assessment Comments:				
	Lecturer Feedback:				
	Input Validation		5		
	Self-assessment Comments:				
	Lecturer Feedback:				
	Script Interaction		3		
	Self-assessment Comments:				
	Lecturer Feedback:				
	User Management		20		
	Self-assessment Comments:				
	Lecturer Feedback:				
	Group Management		5		
	Self-assessment Comments:				
	Lecturer Feedback:				
	Shared Folder Configuration		15		
	Self-assessment Comments:				
	Lecturer Feedback:				

[*] Tick (✓) if completed, circle (○) if some completed, and cross out (×) if not attempted. You can leave comments in the field below (e.g. aspects you did not finish, etc.).

	Activity		Points	Completed[*]	Result
	Link Creation		3		
	Self-assessment Comments:				
	Lecturer Feedback:				
	Alias Management		5		
	Self-assessment Comments:				
	Lecturer Feedback:				
Task 2	***Handling Script Input***		5		
	Self-assessment Comments:				
	Lecturer Feedback:				
	Create Compressed Archive		5		
	Self-assessment Comments:				
	Lecturer Feedback:				
	SCP Transfer		5		
	Self-assessment Comments:				
	Lecturer Feedback:				
	Script Output		4		
	Self-assessment Comments:				
	Lecturer Feedback:				

[*] Tick (✓) if completed, circle (○) if some completed, and cross out (×) if not attempted. You can leave comments in the field below (e.g. aspects you did not finish, etc.).

	Activity	Points	Completed[*]	Result
Formal Aspects	***Documentation***	8		
	Self-assessment Comments:			
	Lecturer Feedback:			
	Code Modularity	5		
	Self-assessment Comments:			
	Lecturer Feedback:			
	Code Commenting	3		
	Self-assessment Comments:			
	Lecturer Feedback:			
	Git Usage	4		
	Self-assessment Comments:			
	Lecturer Feedback:			

[*] Tick (✓) if completed, circle (○) if some completed, and cross out (×) if not attempted. You can leave comments in the field below (e.g. aspects you did not finish, etc.).