Atelier Bash

IFT-2001: Systèmes d'exploitation
GLO-2001: Systèmes d'exploitation pour l'ingénierie

1 Installation

Cet atelier nécessite l'utilisation d'une machine virtuelle du cours, que vous pouvez télécharger à partir de ce lien. Une fois que vous avez téléchargé la machine virtuelle, vous pouvez l'ouvrir à l'aide de VMware ou de VirtualBox. Pour vous authentifier sur la machine virtuelle, utilisez le nom d'utilisateur `glo2001` et le mot de passe `glo2001`.

Une fois authentifié, ouvrez un terminal avec `CTRL-ALT-T` où vous pouvez exécuter des commandes. Pour vérifier que la machine virtuelle fonctionne correctement, exécutez la commande suivante dans le terminal :

```
echo 'Hello world'
```

Si tout est configuré correctement, vous devriez voir s'afficher dans la console le texte `Hello world`. Pour activer le partage du presse-papier (*clipboard*), modifiez l'option dans `Devices -> Shared Clipboard -> Bidirectional`.

2 Consignes

La machine virtuelle fournie est déjà configurée correctement pour les ateliers. Les commandes de corrections sont de la forme `correction_nn`, où `nn` est un nombre entier représentant le numéro de l'exercice, par exemple `correction_03` pour l'exercice 3. Les instructions pour valider les commandes seront fournies avec la première question ayant un résultat attendu.

Pour réaliser cet atelier, vous devrez effectuer chaque exercice directement dans le terminal. Vous devrez utiliser l'éditeur de texte `nano` pour modifier les scripts demandés. Pour ouvrir un fichier avec nano, exécutez la commande suivante dans le terminal.

```
nano test.sh
```

Vous pourrez alors éditer le fichier, inscrivez y

```
echo 'Hello world'
```

Pour quitter nano, appuyez sur `CTRL-X`, puis répondez `y` pour sauvegarder les modifications. Exécutez les commandes suivantes pour valider que le tout fonctionne :

```
chmod +x test.sh
./test.sh
```

Ces commandes devraient afficher le texte `Hello world`.

Pour les aventuriers seulement : Si vous êtes à l'aise avec la ligne de commande, nous vous encourageons à essayer de réaliser cet atelier en utilisant `vim` comme éditeur de texte. Une brève introduction aux commandes de `vim` est disponible sur le site de The Missing Semester par le MIT. `vim` offre de nombreux raccourcis pour éditer du texte et du code de manière très efficace. En tant que programmeur, vous passerez beaucoup de temps à écrire du code, il est donc judicieux d'investir dans l'apprentissage de `vim` pour le reste de votre carrière. Il existe des *plugins* émulant les commandes `vim` pour la majorité des IDEs : Vim pour vscode ou IdeaVim pour les produits JetBrains **Veuillez noter qu'aucune assistance concernant `vim` ne sera fournie pendant cet atelier.**

3 Solution

Les solutions proposées aux exercices de ces ateliers sont disponibles dans le dépôt GitHub suivant : ulavalIFTGLOateliers/IFT2001-Scripting.

4 Bash

C'est une journée comme les autres au bureau. Vous arrivez tôt le matin, votre café à la main et votre ordinateur portable sous le bras. Alors que vous entrez dans votre département, vous remarquez que votre patron, habituellement calme et détendu, marche nerveusement dans le couloir. Son expression livide et ses mains tremblantes attirent immédiatement votre attention.

« Inquiétant, que se passe-t-il ? » lui demandez-vous, soucieux.

« La production a planté », répond-il d'une voix tremblante. « Nous recevons des centaines d'appels de clients, plus rien ne fonctionne... »

Votre cœur s'accélère alors que vous réalisez l'ampleur du problème. Votre entreprise dépend d'un serveur qui héberge la majeure partie de ses services, et il semble qu'il y ait un dysfonctionnement. Votre patron vous fixe droit dans les yeux.

« Résoudre ce problème est votre priorité », vous dit-il avec fermeté.

Il vous tend une feuille de papier avec les identifiants de connexion de l'utilisateur `glo2001` (dont le mot de passe est aussi `glo2001`...) pour accéder au serveur. Vous vous sentez à la fois nerveux et excité à l'idée de résoudre ce problème critique. Votre patron vous conduit alors à la salle des serveurs, où un vieil écran cathodique et un clavier vous attendent. Il vous explique qu'il s'agit d'un système d'exploitation Linux en version serveur, donc sans interface graphique. Vous devrez donc résoudre les problèmes en utilisant uniquement le terminal.

4.1 La ligne de commande et le manuel

Ouvrez un terminal avec le raccourci `CTRL-ALT-T`, ou appuyez sur la touche `super` (la touche Windows sur le clavier) et cherchez pour l'application `Terminal`.

Comme première étape, vous décidez d'inspecter les fichiers présents sur le serveur afin de trouver les fichiers de log. Cependant, vous n'êtes pas certain de la commande à utiliser pour cela.

Heureusement, la ligne de commande vous offre la possibilité de connaître le fonctionnement de chaque commande. La commande `man` permet de lire la page du manuel correspondant à une commande spécifique. Par exemple, pour obtenir la documentation sur la commande `ls`, vous pouvez utiliser la commande suivante :

```
man ls
```

Une fois que vous avez ouvert la page du manuel, vous pouvez naviguer à l'aide des flèches du clavier pour lire le contenu. Pour quitter la page du manuel, appuyez simplement sur la touche `q`.

Aussi, certaines commandes acceptent l'argument `--help` qui affiche un message d'aide décrivant les arguments que l'on peut passer à la commande. Par exemple :

```
ls --help
```

En exécutant cette commande, vous obtiendrez un message d'aide détaillant les différentes options et arguments que vous pouvez utiliser avec la commande `ls`.

N'hésitez pas à utiliser l'argument `--help` avec les commandes que vous souhaitez explorer afin d'obtenir des informations supplémentaires sur leur utilisation. Cela peut vous aider à mieux comprendre les fonctionnalités disponibles et à utiliser correctement les commandes dans votre exploration du serveur.

Pour certaines commandes comme `cd` (qui est une commande incluse directement dans le *Shell Bash*), il faut plutôt utiliser

```
help cd
```

NOTEZ BIEN: Raccourcis dans le terminal

Auto complétion : Vous pouvez utiliser la touche `TAB` pour obtenir de l'auto complétion dans le terminal.

Quitter une commande : Dans un terminal, le raccourcit `CTRL-C`, plutôt que de copier, termine l'exécution d'une commande. Par exemple, la commande `yes` répète indéfiniment la lettre `y` dans le terminal. Pour quitter, appuyer sur `CTRL-C` :

```
yes
# CTRL-C pour quitter l'execution de yes
```

Pour copier et coller, le terminal utilise plutôt `CTRL-SHIFT-C` et `CTRL-SHIFT-V`
Pour fermer un terminal, vous pouvez utiliser `CTRL-D`.

Exercice 1: Commandes de base

Cet exercice vise à vous familiariser avec l'utilisation de quelques commandes de base qui vous seront utiles tout au long de l'atelier. Pour ce faire, utilisez la commande `man` et l'argument `--help` pour obtenir des informations détaillées sur le fonctionnement de chaque commande. Dans le fichier `exercice_01.txt`, décrivez brièvement l'utilité de chacune des commandes suivantes. Cet aide mémoire vous sera utile tout au long de l'atelier. Vous pourrez le consulter avec la commande `cat exercice_01.txt`.

Ouvrez `exercice_01.txt` avec `nano` avec :

```
nano exercice_01.txt
```

Dans un autre terminal (`CTRL-ALT-T` pour ouvrir un autre terminal ou `CTRL-SHIFT-T` pour ouvrir un autre terminal dans un onglet), utilisez le terminal pour déterminer le comportement des commandes suivantes :

Commandes
`nano`
`cd`
`ls`
`cat`
`mkdir`
`rm`
`rmdir`
`mv`
`pwd`
`cp`
`chmod`
`touch`

Solution: 1

Commande	Définition
`nano`	Éditeur de texte simple
`cd`	Permet de changer de dossier courant
`ls`	Permet de lister les fichiers dans un dossier
`cat`	Affiche le contenu d'un fichier
`mkdir`	Créé un dossier
`rm`	Supprime un fichier ou un dossier
`rmdir`	Supprime un dossier vide
`mv`	Déplacer un fichier
`pwd`	Affiche le chemin du répertoire courant
`cp`	Copie un fichier
`chmod`	Change les permissions d'un fichier
`touch`	Met à jour le timestamp d'un fichier, ou le crée s'il n'existe pas

Maintenant que vous êtes familiarisé avec quelques commandes de base, il est temps d'inspecter les fichiers du serveur et trouver les fichiers de log du serveur.

Exercice 2: Navigation du système de fichiers

Utilisez les commandes listées plus haut afin d'explorer l'arborescence de dossier et trouver le fichier avec l'extension `.log`.

Notez bien: Voici quelques chemins spéciaux
- `.` représente le dossier courant ;
- `..` représente le dossier parent ;
- ∼ représente le dossier `home` de l'utilisateur (`/home/glo2001/`) ;
- `-` représente le chemin du dernier dossier visité.

Vous pouvez utiliser ces chemins spéciaux avec plusieurs commandes, notamment `cd`.

```
# Navigue dans le dossier abc/def
cd abc/def

# Navigue dans le dossier parent (abc)
cd ..

# Retourne au dossier home
cd
# ou
cd ~

# Retourne au dossier precedant (abc)
cd -
```

1. Listez les fichiers dans le répertoire courant
2. Déplacez-vous dans les différents dossiers et tentez de trouver le fichier portant l'extension `.log`
3. Copiez le chemin **absolu** (depuis la racine du système de fichier `/`) du dossier dans lequel se trouvent les fichiers de log dans le fichier ∼`/out_02.txt`. Ne pas ajouter de nouvelle ligne à la fin.

Exécutez la commande de correction `correction_02.sh` pour valider votre réponse.

Solution: 2 Voici les commandes à exécuter pour réaliser l'exercice

1. `ls`
2. `cd ApplicationData/output/logs`
3. `ls`
4. `pwd`

Le chemin attendu est `/home/glo2001/ApplicationData/output/logs`

Maintenant que vous avez trouvé le fichier de log, il est temps d'en faire une copie dans votre dossier `home`.

Notez bien: Le dossier home est l'endroit où sont stockés les fichiers personnels d'un utilisateur. Chaque utilisateur a son propre dossier dans `/home/`. Dans votre cas, votre utilisateur a comme nom d'utilisateur `glo2001`, alors sont dossier home est situé à `/home/glo2001`.
Il existe aussi un raccourcit pour référer au dossier home ∼. Ainsi, chacune des commandes suivantes vous permet de retourner au dossier home

```
# Utilisation de chemin absolu
cd /home/admin
# Utilisation de tilde
cd ~
# Sans argument, cd retourne au dossier home
cd
```

Exercice 3: Dossier `home` et copie

Copiez le fichier `.log` dans le dossier `~/log_backup`.

1. Retournez dans votre dossier `home` ;
2. Créez un nouveau dossier qui s'appelle `log_backup` dans votre dossier `home` ;
3. Copiez le fichier .log dans le dossier backup, en lui donnant le nom `build_backup.log`.

Exécutez la commande de correction `correction_03.sh` pour valider votre réponse.

Solution: 3 Liste des commandes pour réaliser l'exercice

1. `cd`
2. `mdkdir log_backup`
3. `cp ~/ApplicationData/output/logs/build.log ~/log_backup/build_backup.log`

Maintenant que vous avez effectué une sauvegarde des logs de votre application, vous décidez d'exécuter le script de diagnostic du projet. Veuillez retourner dans le répertoire du fichier log et tenter d'exécuter le script `./diagnostic.sh`.

Cependant, vous rencontrez une erreur de permissions. Pour résoudre ce problème, inspectez les permissions du script en utilisant la commande `ls -l`. Ensuite, modifiez les permissions pour rendre le script exécutable.

Notez bien: `chmod`, abréviation de *change mode*, est une commande utilisée pour modifier les permissions d'accès aux fichiers et répertoires. Les permissions en bash se réfèrent aux droits d'accès accordés aux utilisateurs et aux groupes pour lire, écrire et exécuter des fichiers et des répertoires. Ces permissions permettent de contrôler qui peut effectuer quelles opérations sur un fichier ou un répertoire donné. Vous pouvez lister les permissions des fichier et des répertoire avec `ls -l`.
Les permissions sont généralement représentées sous la forme d'un caractère suivit de trois groupes, chacun ayant trois caractères, soit un total de neuf caractères, affichés dans un ordre spécifique :

1. Le premier caractère indique s'il s'agit d'un fichier (`-`) ou d'un répertoire (`d` pour *directory*).
2. Le premier groupe de trois caractères représente les permissions du propriétaire du fichier.
3. Le deuxième groupe de trois caractères représente les permissions du groupe auquel appartient le fichier.
4. Le troisième groupe de trois caractères représente les permissions pour les autres utilisateurs.

Chaque groupe de trois caractères se compose des trois types de permissions suivants :

1. `r` (*read*) : Permet de lire le contenu du fichier ou du répertoire.
2. `w` (*write*) : Permet de modifier ou de supprimer le fichier (ou le contenu du répertoire).
3. `x` (*execute*) : Permet d'exécuter un fichier (ou de parcourir un répertoire).

Pour modifier ces permissions, vous pouvez utiliser la commande `chmod`. Par exemple, pour ajouter (`+`) la permission d'exécution à un fichier, vous pouvez utiliser la commande suivante :

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

Pour retirer la permission d'écriture à un répertoire, vous pouvez utiliser la commande suivante :

```
chmod -w dossier
```

Exercice 4: Permissions

Modifiez les permissions du script `diagnostic.sh`, puis exécutez le script.

- Inspecter les permissions du fichier ;
- `chmod` avec l'argument `+x` pour modifier les permissions ;
- Exécuter `diagnostic.sh`.

Exécutez la commande de correction `correction_04.sh` pour valider votre réponse.

Solution: 4

```
# On regarde les permissions
ls -l
# On change les permissions
chmod +x diagnostic.sh
# On execute le script
./diagnostic.sh
```

Le script a généré une dizaine de fichiers `.out` contenant les résultats de l'analyse du système. Vous devez déplacer ces fichiers dans un nouveau dossier nommé `output`. Au lieu de déplacer chaque fichier manuellement avec la commande `mv out_01.out output/`, ce qui serait fastidieux, vous pouvez utiliser le caractère générique *wildcard* (`*`), qui permet de sélectionner plusieurs fichiers à la fois.

Avant de déplacer les fichiers, vous pouvez essayer la commande `cat *`. Cette commande affiche le contenu de tous les fichiers présents dans le répertoire courant.

Cependant, dans votre cas, vous ne souhaitez pas sélectionner tous les fichiers. Vous pouvez spécifier un format spécifique en ajoutant un préfixe ou un suffixe aux noms des fichiers. Par exemple, pour sélectionner tous les fichiers `.sh`, vous pouvez utiliser la commande `ls *.sh`. Cela affichera la liste des fichiers portant l'extension `.sh`.

Exercice 5: Wildcards

- Créez un dossier nommé `output_backup` dans le dossier `~/ApplicationData/output/logs` ;
- Déplacez tous les fichiers terminant par `.out` dans le dossier en une seule commande.

Exécutez la commande de correction `correction_05.sh` pour valider votre réponse.

Solution: 5

```
# Deplacer dans le bon dossier
cd ~/ApplicationData/output/logs
# Creation du dossier output
mkdir output_backup
# Deplacer les fichiers dans le dossier
mv *.out output_backup/
```

En plus de générer des fichiers .out, le script a également généré des fichiers temporaires `.tmp` et un dossier nommé `temp`. Ces fichiers et ce dossier peuvent être supprimés.

Exercice 6: Suppression de fichiers et de dossiers

- Supprimez les fichiers ayant l'extension `.tmp` ;
- Supprimez le dossier `temp`.

Exécutez la commande de correction `correction_06.sh` pour valider votre réponse.

Solution: 6

```
# Suppression des fichiers .tmp
rm *.tmp
# Suppression du dossier
rm -r temp
```

4.2 Scripts

Pour simplifier la tâche de déplacement et de suppression des fichiers générés par le script `diagnostic.sh`, vous pouvez créer un script qui automatisera ces actions pour vous.

Notez bien: Voici un exemple de script Bash

```
#!/usr/bin/env bash
echo 'Debut du script'
ls *.sh
echo 'Fin du script'
```

Dans ce script, la ligne `#!/usr/bin/env bash` est appelée un *shebang* (she = `#`, bang = `!`). Elle indique à l'interpréteur quel programme doit être utilisé pour exécuter le script, dans ce cas, il s'agit de Bash. Il serait aussi possible de spécifier un autre programme comme interpréteur. Par exemple, pour interpréter le script comme du Python3, on utiliserait le shebang suivant : `#!/usr/bin/env python3`.

Utiliser `#!/usr/bin/env bash` est généralement préférable à `#!/bin/bash`, car cela permet de rechercher l'emplacement de l'exécutable Bash dans l'environnement de l'utilisateur, ce qui le rend plus portable d'un système à l'autre.

Vous pouvez créer un fichier texte avec l'extension `.sh` (par exemple, `exemple.sh`), y copier le script ci-dessus, puis rendre le fichier exécutable à l'aide de la commande `chmod +x exemple.sh`. Ensuite, vous pourrez exécuter le script en utilisant `./exemple.sh` pour exécuter le script.

Exercice 7: Scripting de base

1. Créer un script nommé `cleanup.sh` dans le dossier `~/ApplicationData/output/logs/` ;
2. Donnez les permissions d'exécution au script ;
3. Utilisez les commandes que vous avez tapées dans les exercices précédents pour créer votre script ;
   - Créez un dossier nommé `output_backup` ;
   - Déplacez tous les fichiers terminant par `.out` dans le dossier en une seule commande ;
   - Supprimez les fichiers ayant l'extension `.tmp` ;
   - Supprimez le dossier `temp`.
4. Assurez-vous de supprimer le dossier `output` que vous avez créé plus tôt ;
5. Exécutez `diagnostic.sh` un autre fois, et tester votre script.

Exécutez la commande de correction `correction_07.sh` pour valider votre réponse.

Solution: 7

- `touch cleanup.sh`
- `chmod +x cleanup.sh`
- `nano cleanup.sh` et y ajouter le texte suivant

```
#!/usr/bin/env bash

# Creation du dossier output_backup
mkdir output_backup
# Deplacer les fichiers dans le dossier
mv *.out output_backup/

# Suppression des fichiers .tmp
rm *.tmp
# Suppression du dossier
rm -r temp
```

4.3 Composition de programmes et commandes avancées

Maintenant que vous avez effectué quelques opérations de nettoyage et que vous vous êtes familiarisé avec la ligne de commande, il est temps d'analyser les logs et les sorties du programme de diagnostic. Pour cela, vous devrez combiner plusieurs commandes en utilisant le `piping` (ou pipeline) et des commandes plus avancées. Avant d'aborder le sujet, vous explorerez quelques commandes plus avancées qui vous seront utiles pour la suite.

4.3.1 Commandes avancées

Exercice 8: Commandes avancées

Comme à l'exercice 1, utilisez la commande `man` et l'argument `--help` pour obtenir des informations détaillées sur les commandes du tableau suivant. Ces commandes sont un peu différentes de celles vues à l'exercice 1, celles-ci peuvent prendre en entrée un fichier, ou on peut y `pipe` la sortie d'une autre commande, ce qui sera le sujet de la section suivante. Vous pouvez créer un fichier de test (`test_file.txt`) afin de tester les commandes.
Dans le fichier `exercice_08.txt`, décrivez brièvement l'utilité de chacune des commandes suivantes. Cet aide mémoire vous sera utile tout au long de l'atelier.

Commandes
`tac`
`less`
`find`
`grep`
`sort`
`uniq`
`wc`
`head`
`tail`
`du`
`curl`
`sed`
`awk`
`kill`
`sleep`

Solution: 8

Commande	Définition
`tac`	Inverse l'ordre des lignes d'un fichier
`less`	Pagination de texte
`find`	Recherche de fichiers
`grep`	Filtre des lignes selon un pattern
`sort`	Trie les lignes
`uniq`	Enlève les lignes doublons
`wc`	Compte le nombre de lignes, de mots et de caractères
`head`	Affiche le début d'un fichier
`tail`	Affiche la fin d'un fichier
`du`	Affiche la taille de fichiers
`curl`	Requête réseau (HTTP, FTP, etc.)
`sed`	Éditeur de ligne
`awk`	Interpréteur pour le langage `awk`
`kill`	Arrête un processus
`sleep`	Attend un nombre de secondes

Avec ces nouvelles connaissances sur les commandes Bash, il est maintenant temps de retourner à votre problème de serveur.

Exercice 9: Commandes avancées 1

Le fichier `messages.txt` dans le dossier de log contient les messages d'erreurs qui causent la panne du système. Malheureusement, ce fichier contient aussi beaucoup de log qui ne vous sont pas utiles. Plutôt que de manuellement lire l'entièreté du fichier, vous décidez d'utiliser les commandes Bash que vous venez de découvrir.

Utilisez une commande afin d'afficher toutes les lignes de `messages.txt` qui contiennent la chaîne de caractère `Error` et copiez le résultat dans le fichier `~/errors.txt`.

Exécutez la commande de correction `correction_09.sh` pour valider votre réponse.

Solution: 9

```
grep Error messages.txt
```

Exercice 10: Commandes avancées 2

Après avoir inspecté les erreurs, que vous avez copiées dans `/errors.txt`, vous réalisez qu'il y a beaucoup de doublons. Utilisez une commande Bash afin d'enlever les lignes dupliquées et copiez le résultat dans `~/errors_2.txt`.

Exécutez la commande de correction `correction_10.sh` pour valider votre réponse.

Solution: 10

```
uniq ~\errors.txt
```

Exercice 11: Commandes avancées 3

Après avoir inspecté les erreurs filtrées du fichier `~/errors_2.txt`, remplacer les erreurs possédant un code 400 (400, 403 et 404) par des avertissement. Avec `sed`, remplacer le texte en utilisant l'expression régulière
`'Error \(4[0-9]\+\)`
par le texte suivant :
`Warning \1'`
où `\1` va copier le nombre capturé dans l'expression régulière.
Indice : la commande aura la forme `sed 's/regex1/regex2/g`. `s` indique qu'il s'agit d'une **s**ubstitution que l'on applique **g**lobalement.
Utilisez une commande Bash afin modifier les messages et copiez le résultat dans `~/errors_3.txt`.

Exécutez la commande de correction `correction_11.sh` pour valider votre réponse.

Solution: 11

```
sed 's/Error \(4[0-9]\+\)/Warning \1/g' ~/errors_2.txt
```

4.3.2 Composition de programme

La composition de programme est au cœur de la philosophie Unix. Voici un extrait de *The Art of Unix Programming* [1] décrivant l'importance de la composition de programme (chapitre complet disponible ici).

> It's hard to avoid programming overcomplicated monoliths if none of your programs can talk to each other.
>
> Unix tradition strongly encourages writing programs that read and write simple, textual, stream-oriented, device-independent formats. Under classic Unix, as many programs as possible are written as simple filters, which take a simple text stream on input and process it into another simple text stream on output.
>
> Despite popular mythology, this practice is favored not because Unix programmers hate graphical user interfaces. It's because if you don't write programs that accept and emit simple text streams, it's much more difficult to hook the programs together.
>
> Text streams are to Unix tools as messages are to objects in an object-oriented setting. The simplicity of the text-stream interface enforces the encapsulation of the tools. More elaborate forms of inter-process communication, such as remote procedure calls, show a tendency to involve programs with each others' internals too much.
>
> To make programs composable, make them independent. A program on one end of a text stream should care as little as possible about the program on the other end. It should be made easy to replace one end with a completely different implementation without disturbing the other.
>
> — Chapter 1. Philosophy, Rule of Composition

Cette composition de programme peut se faire à l'aide de différents opérateurs, qui seront l'objet des prochaines sections.

Composition L'opérateur ; permet d'exécuter une commande après l'autre sur une même ligne.
(`commande1 ; commande2`)
Par exemple, pour l'exercice 7, on pourrait réécrire le programme comme suit :

```
mkdir output; mv *.out output/; rm *.tmp; rm -r temp
```

L'opérateur && permet de ne réaliser la deuxième commande que si la première a réussit (retourne un code de sortie égal à zéro).
(`commande1 && commande2`)

```
# Affiche "Hello World" car les deux commandes reussisent
echo 'Hello' && echo 'World'

# On affiche hello car le dossier existe, donc cd reussit
cd dossier_qui_existe && echo 'Hello

# On n'affiche pas Hello car le dossier n'existe pas, donc cd echoue
cd dossier_non_existant && echo 'Hello'
```

L'opérateur || permet de ne réaliser la deuxième commande que si la première a échoué (retourne un code de sortie différent de zéro).
(`commande1 || commande2`)

```
# Affiche seulement Hello, car echo 'Hello' reussit
echo 'Hello' || echo 'World'

# On n'affiche pas Hello, car cd reussit
cd fichier_qui_existe || echo 'Hello'

# On affiche hello, car cd echoue
cd fichier_non_existant || echo 'Hello'
```

Tuyaux L'opérateur | permet de passer la sortie d'une commande en entrée à une autre. Cet opérateur est aussi appelé *pipe*.
(`commande1 | commande2`)

```
# Liste les fichiers, et ne conserve que les .txt
ls | grep ".txt"

# Trie les lignes d'un fichier, ne conserve que les 5 premieres lignes
cat file.txt | sort | head -n 5
```

Redirection L'opérateur > redirige la sortie d'une commande vers un fichier, écrasant le fichier
(`commande > fichier`)

```
ls > file.txt
```

Par exemple, il est possible de télécharger un fichier texte avec `curl` :

```
curl https://www.ulaval.ca/ > ulaval.txt
```

Pour ignorer la sortie d'une commande :

```
ls > /dev/null 2>&1
```

L'opérateur < redirige l'entrée d'une commande depuis un fichier.
(`commande < fichier`)

```
wc < file.txt
```

L'opérateur >> comme > permet de rediriger la sortie d'une commande vers un fichier, mais ajoute à la fin du fichier (*append*) plutôt que de l'écraser :
(`commande >> fichier`)

```
ls >> file.txt
```

L'opérateur << comme <, mais permet de passer plusieurs lignes.
(`commande >>delim [plusieurs lignes] delim`)

```
cat <<EOF
    abc
    def
    hij
    EOF
# On peut remplacer EOF (end of file) par tout autre chaine de charactere
cat <<ABC
    abc
    def
    hij
    ABC
```

Exemples Voici quelques exemples utilisant les tuyaux et les redirections.

```
# Trouve les 5 premiers .txt en ordre alphabetique
ls -l | grep ".txt" | sort | head -n 5
# En ordre, voici ce que la commande fait
# 1. Liste les fichiers
# 2. Ne garde que les fichiers contenant .txt dans leur nom
# 3. Trie selon l'ordre alphabetique
# 4. Ne conserve que les 5 premiers resultats

# Ne conserve que les lignes contenant le texte "warning"
# remplace "warning" par "error", puis ecrit le resultat dans output.txt
cat data.txt | grep "warning" | sed 's/warning/error/g' > output.txt
# En ordre, voici ce que la commande fait
# 1. Affiche le contenu de data.txt
# 2. Filtre les lignes afin de ne conserver que les lignes contenant "warning"
# 3. Applique un regex pour remplacer "warning" par "error"
# 4. > permet d'ecrire le resultat dans le fichier output.txt

# Genere un fichier de test
echo -e "1\t2\n2\t3\n3\t4\n" > foo.txt
# En ordre, voici ce que la commande fait
# 1. Affiche une chaine de caractere formatee
# (l'argument -e fait en sorte que \t sera interprete comme une tabulation, \n comme un
    ↪ retour a la ligne)
# 2. Ecrit le resultat dans foo.txt

# Calcule la somme de chacune des colones
cat foo.txt | awk '{sum += $1; sum2 += $2} END {print sum; print sum2}' \
    | xargs echo "Sum of both columns"
# En ordre, voici ce que la commande fait
# 1. Affiche le contenu de foo.txt
# 2. Awk
# a. Cree une variable sum, a laquelle on ajoute la valeur
# de la premiere colonne ($1) pour chaque ligne
# b. Cree une variable sum2 a laquelle on ajoute la valeur
# de la deuxieme colonne ($2) pour chaque ligne.
# c. Affiche les valeurs de sum et sum2
# d. \ permet de continuer la commande sur la ligne suivante
# 3. Passe le resultat de awk a la commande echo qui va afficher la somme
```

On vous recommande d'essayer ces commandes une à la fois afin de bien comprendre chaque étape du pipeline, par exemple :

```
ls -l
ls -l | grep ".txt"
ls -l | grep ".txt" | sort |
ls -l | grep ".txt" | sort | head -n 5
```

Après cet interlude sur les commandes Bash et le piping, il est temps de régler le problème du serveur.

Exercice 12: Piping et redirection

Utilisez le piping et la redirection pour réécrire les exercices 9, 10 et 11 en une seule commande.

Écrivez le résultat de votre script dans le fichier `~/out_12.txt` (avec une redirection).

Exécutez la commande de correction `correction_12.sh` pour valider votre réponse.

Solution: 12

```
grep Error messages.txt | uniq | sed 's/Error \(4[0-9]\+\)/Warning \1/g' > ~/out_12.txt
```

Les messages d'erreur semblent un problème avec un fichier trop gros, utilisez vos connaissances en Bash pour trouver les plus gros fichiers dans le répertoire `files`.

Exercice 13: Taille de fichiers

Afin de trouver les cinq plus gros fichiers du dossier `Documents`, utilisez cette commande qui retourne la liste des fichiers ainsi que leur taille.

```
find Documents -type f | xargs du -sb
```

Chaque ligne contient la taille en octets et le nom du fichier séparé par un caractère `TAB`.

Utilisez la sortie de cette commande afin de trouver les cinq fichiers les plus volumineux. Votre script doit retourner cinq lignes dans le même format que `du`, c'est-à-dire que chaque ligne doit avoir le format suivant :

```
taille-en-octets nom-du-fichier
```

Écrivez le résultat de votre script dans le fichier `~/out_13.txt` (avec une redirection).

Exécutez la commande de correction `correction_13.sh` pour valider votre réponse.

Solution: 13

```
find ~/Documents -type f | xargs du -sb | sort -rh | head -n 5 > ~/out_13.txt
# ou
find ~/Documents -type f | xargs du -sb | sort -h | tac | head -n 5 > ~/out_13.txt
# ou
find ~/Documents -type f | xargs du -sb | sort -h | tail -n 5 | tac > ~/out_13.txt
# ou
find ~/Documents -type f -exec du -sb {} + | sort -rh | head -n 5 > ~/out_13.txt
```

Exercice 14: Analyse de données

Le fichier ~/ApplicationData/db.tsv contient des données sous la forme de TSV (*TAB separated value*). Voici le format du fichier :

id	date	name	type	size
123	2023-12-25	foo	error_log	23

Utilisez ce fichier afin de trouver les 10 lignes avec la plus petite taille (colonne size), ne conserver que la colonne nom. Vous devez conserver l'entête de la colonne conservée, c'est-à-dire la première ligne de ~/out_14.txt devrait être `name`.

Notez bien: Afin de réaliser cette tâche, vous pouvez utiliser `awk`, un langage de programmation spécialisé pour la manipulation de texte. Par exemple, pour extraire la 1e et la 3e colonne, séparé par un `TAB`, vous pouvez utiliser la commande.

```
awk '{print $1 "\t" $3}' file.txt
```

Adaptez cette ligne afin de résoudre le problème.

Écrivez le résultat de votre script dans le fichier ~/out_14.txt (avec une redirection).

Exécutez la commande de correction `correction_14.sh` pour valider votre réponse.

Solution: 14

```
awk '{print $5 " " $3}' db.tsv | sort -h | head -n 11 | awk '{print $2}' > ~/out_14.txt
# ou
awk '{print $5 " " $3}' db.tsv | sort -h | head -n 11 | sed 's/^[^ ]* //' > ~/out_14.txt
```

4.4 Scripting

Maintenant que vous êtes plus à l'aise avec Bash, il est temps de créer des scripts plus avancés.

NOTEZ BIEN:

```bash
#!/usr/bin/env bash
# Les arguments du script sont disponibles avec $n (ou n est un nombre naturel)
echo "The script name is: $0"
echo "The first argument is: $1"
echo "The second argument is: $2"
echo "All arguments:$@"
# {@:2:3} veut dire prendre une slice commencant a 2 (1-based) de longueur 3
echo "Arguments from 2 to 4: ${@:2:3}"

# Variables, noter qu'il n'y a pas d'espace autour du signe =
name="John"
age=25
# $age permet de remplacer la valeur de la variable dans la chaine de caractere entre "
echo "My name is $name and I am $age years old."
# Variables d'environnement
echo "The value of HOME is: $HOME"

# Substitution
current_directory=$(pwd)
# Les substitutions fonctionnent seulement pour des double quotes "
echo "The current directory is: $current_directory using double quotes"
# Pas pour des single quotes '
echo 'The current directory is: $current_directory using single quotes'
# Il est aussi possible d'appeler une commande directement dans une chaine de caractere
path="$(pwd)"/my/path
echo "Path: $path"

# Conditionnels
if [ "$age" -ge 18 ]; then
  echo "You are an adult."
else
  echo "You are a minor."
fi

# Boucles
for i in 1 2 3 4 5
do
    echo $i
done

fruits=("apple" "banana" "orange")
for fruit in "${fruits[@]}"
do
    echo "I like $fruit"
done

counter=1
while [ $counter -le 5 ]
do
    echo $counter
    ((counter++))
done

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

greet "Alice"
```

Finalement, après tout ce temps à déboguer, vous décidez qu'il serait plus sage de revenir à une version précédente. Pour ce faire, vous décidez d'écrire un script qui va essayer différentes version du projet, en commençant

par la plus récente, et en reculant jusqu'à ce que le script de test passe. Ce script sera réalisé en plusieurs étapes.

Exercice 15: Script de test

Complétez le script suivant afin de tester que le serveur Python est fonctionnel. Sauvegardez ce script dans le fichier ~/`test.sh`, et donnez-lui les permissions d'exécution.

```
#!/usr/bin/env bash

cd ~/server

# Lance le serveur en arriere plan
python3 -m uvicorn main:app &
# Sauvegarde le process ID (pid) du serveur avec $!
# $! retourne le pid du dernier processus lance en arriere plan
server_pid=$!

# La fonction ‘stop_server‘ arrete le processus du serveur
stop_server() {
  kill $server_pid
}

# Enregistre la fonction ‘stop_server‘ qui sera appelee quand ce script terminera
trap stop_server EXIT

# Attendre que le server soit pret
sleep 1

# TODO faire une requete HTTP a l'adresse localhost:8000

# TODO Si le resultat est une erreur, quitte avec un code 1 avec la commande ‘exit 1‘
# $? contient le code de retour de la derniere commande
```

Exécutez la commande de correction `correction_15.sh` pour valider votre réponse.

Solution: 15

```
#!/usr/bin/env bash

cd ~/server

# Lance le serveur en arriere plan
python3 -m uvicorn main:app &
# Sauvegarde le process ID (pid) du serveur avec $!
server_pid=$!

# La fonction ‘stop_server‘ arrete le processus du serveur
stop_server() {
  kill $server_pid
}

# Enregistre la fonction ‘stop_server‘ qui sera appelee quand ce script terminera
trap stop_server EXIT

# Attendre que le server soit pret
sleep 1

# TODO faire une requete HTTP a l’adresse localhost:8080
curl localhost:8000 > /dev/null 2> /dev/null

# TODO Si le resultat est une erreur, quitte avec un code 1 avec la commande ‘exit 1‘
# $? contient le code de retour de la derniere commande
if [ $? -eq 0 ]; then
  echo "Server is up"
else
  echo "Server is down"
  exit 1
fi
```

Exercice 16: Script de retour de version

Complétez le script suivant afin de retourner à une version précédente du serveur jusqu'à ce que le script de test passe. Sauvegardez ce script dans le fichier ∼/`revert.sh`, et donnez lui les permissions d'exécution.

```
#!/usr/bin/env bash

cd ~/server

revert_last_commit() {
  git reset HEAD~1 --hard
}

exit_status=1
# TODO faire une boucle while tant que $exit_status n’est pas egal a 0
# A chaque iteration, executez le script test.sh
# Si le script termine avec un code 0, arreter le script exit 0
# Sinon, appelez la fonction ‘revert_last_commit‘
```

Exécutez votre script.
Exécutez la commande de correction `correction_16.sh` pour valider votre réponse.

Solution: 16

```
#!/usr/bin/env bash

cd ~/server

revert_last_commit() {
  git reset HEAD~1 --hard
}

exit_status=1
while [ $exit_status -ne 0 ]; do
  # Execute le script de test
  ~/test.sh > /dev/null 2>&1
  # Sauvegarde le status du script de test
  exit_status=$?

  if [ $exit_status -ne 0 ]; then
    echo "The script returned a non-zero exit status ($exit_status). Reverting commit and
        ↪ retrying..."
    revert_last_commit
  else
    echo "The script returned 0 (success)."
    exit 0
  fi

  # Add a delay between retries (optional)
  sleep 1
done
```

4.5 Extras

4.5.1 .bashrc

`.bashrc` est un fichier exécuté lors de l'ouverture d'un terminal. Voici quelques utilisations courantes :

- Configuration par utilisateur de la Shell ;
- Définir des alias (le sujet de la prochaine section) ;
- Configurer des variables d'environnement (par exemple modifier la variable `PATH`, sujet d'une section extra) ;
- Configuration du prompt (ce qui est affiché dans la Shell `glo2001@server:~$`) grâce à la variable `PS1`, par exemple pour afficher le nom de la branche Git ;
- Définir des fonctions personnalisées.

4.5.2 Alias

Il est possible de définir des alias pour des commandes communes. Ces commandes sont souvent ajoutées au `.bashrc`. Voici quelques exemples d'alias :

```
# Affiche les permissions
alias ll='ls -alF'
# Affiche les fichier caches (commencant par un point)
alias la='ls -A'

# Raccourcits pour apt
alias update='sudo apt update'
alias upgrade='sudo apt upgrade'
alias install='sudo apt install'

# Git aliases
alias ga='git add'
alias gs='git status'
alias gc='git commit'
alias gp='git push'
```

```
# Raccourcits vers des dossiers commins
alias docs='cd ~/Documents'
alias dl='cd ~/Downloads'
alias desk='cd ~/Desktop'

# Une petite blague
alias emacs='vim'
```

4.5.3 PATH

La variable `PATH` est une variable d'environnement utilisée par le Shell pour déterminer les répertoires dans lesquels il recherche les exécutables lorsque vous tapez une commande dans le terminal. Lorsque vous entrez une commande, le Shell va parcourir chaque répertoire spécifié dans la variable `PATH` de gauche à droite, cherchant un fichier exécutable portant le nom de la commande. Le premier fichier exécutable trouvé est alors exécuté.

La valeur de la variable `PATH` est une liste de répertoire, séparés par des deux-points (`:`). Ajouter un chemin à cette variable permet d'accéder rapidement à des commandes sans devoir spécifier le chemine complet de chaque exécutable. Par exemple, pour ajouter un répertoire à la variable `PATH` :

```
export PATH=$PATH:~/.bin
```

4.5.4 Alias Git

Les alias Git sont des raccourcis personnalisés que vous pouvez créer pour simplifier l'utilisation des commandes Git fréquemment utilisées. Les alias vous permettent de définir des commandes abrégées qui exécutent une séquence de commandes Git plus longue ou complexe. Vous pouvez ajouter des commandes dans le fichier `.gitconfig` qui se trouve dans votre dossier `home`. Voici un exemple d'alias Git, extrait de willGuimont/.gitconfig.

```
[alias]
	lg1 = log --graph --abbrev-commit --decorate --format=format:'%C(bold blue)%h%C(
	    ↪ reset) - %C(bold green)(%ar)%C(reset) %C(white)%s%C(reset) %C(dim white)- %an%
	    ↪ C(reset)%C(bold yellow)%d%C(reset)' --all
	lg2 = log --graph --abbrev-commit --decorate --format=format:'%C(bold blue)%h%C(
	    ↪ reset) - %C(bold cyan)%aD%C(reset) %C(bold green)(%ar)%C(reset)%C(bold yellow)
	    ↪ %d%C(reset)%n'' %C(white)%s%C(reset) %C(dim white)- %an%C(reset)' --all
	lg = !"git lg1"
	st = status
	co = commit
	com = commit -m
	coam = commit --amend
	a = add
	aa = add .
	p = push
	pu = push -u origin HEAD
	cb = checkout -b
	c = checkout
	m = merge
	f = fetch
	g = pull
	wip = commit -a -m "wip"
	wap = commit -a -m
	todo = !"git commit -a -m \"todo\" && git p"
	refs = !"git pull && git commit -a -m \"updated references\" && git p"
	figs = !"git pull && git commit -a -m \"updated figures\" && git p"
	wdiff = diff --word-diff-regex=.
```

Références

[1] Eric Steven Raymond. *The Art of Unix Programming*. Addison-Wesley, Boston, 2004. ISBN 0131429019 9780131429017. URL http://www.faqs.org/docs/artu/.