Système - TP1
Bash et bases d'administration système

1 Quelques rappels pratiques

Lorsque vous êtes confrontés à une commande que vous ne connaissez pas, tapez **man [nom de la commande]** dans le terminal pour lire la page de manuel de cette commande. Exemple : **man ls**. Certaines commandes n'ont pas de page de manuel : vous pouvez alors essayer de lancer la commande avec l'option **--help** pour obtenir de l'aide. Exemple : **ls --help**.

Dans un terminal, vous pouvez utiliser l'autocomplétion avec la touche **Tab**. Par exemple si vous tapez **ls /et** puis **Tab**, vous devriez voir apparaître **ls /etc/**.

Si plusieurs choix sont possibles, il faut appuyer une seconde fois sur **Tab** pour faire apparaître les options disponibles. Par exemple si vous tapez **ls /b** puis **Tab**, rien ne se passe car plusieurs des dossiers contenus dans `/` commencent par `b`, et le terminal ne peut pas savoir lequel vous cherchez. Si vous appuyez une seconde fois sur **Tab**, le terminal vous informe que `bin/` et `boot/` sont disponibles. Vous pouvez alors taper **o** puis **Tab**, et le terminal complètera votre commande en **ls /boot/**.

Prenez l'habitude d'utiliser l'autocomplétion, taper vos noms de fichiers vous-même est beaucoup trop long !

Pour utiliser les efforts, vous pouvez également utiliser l'historique du terminal. Appuyez sur les touches **Flèche haut** et **Flèche bas** pour parcourir les commandes précédemment exécutées, sans avoir à les retaper. Vous pouvez aussi utiliser **Ctrl+R** pour effectuer une recherche dans les commandes précédentes.

Lorsqu'une commande est en cours d'exécution dans le terminal courant, appuyez sur **Ctrl+C** pour interrompre l'exécution.

Pour copier-coller dans un terminal, utilisez **Shift+Ctrl+C** pour copier et **Shift+Ctrl+V** pour coller. Si vous souhaitez copier-coller une commande depuis un fichier pdf, il faut donc faire **Ctrl+C** dans le pdf (puisque vous êtes hors du terminal), puis **Shift+Ctrl+V** dans le terminal.

Ctrl+A permet de placer le curseur au début de la ligne courante, et **Ctrl+E** permet de le replacer en fin de ligne.

2 Redirections

En bash, les symboles `>`, `>>`, `<`, `<<` et `<<<` permettent de rediriger les entrées et sorties d'une commande.

On présume que les fichiers `f1` et `f2` n'existent pas encore, puis on exécute les commandes suivantes.

```
$ echo "Premier test f1" > f1
$ echo "Second test f1" > f1
$ echo "Premier test f2" >> f2
$ echo "Second test f2" >> f2
```

Question 1. Quel est le contenu des deux fichiers après l'exécution de ces commandes ?

Testez les commandes suivantes :

```
$ rmdir /etc
$ rmdir /etc > f3
$ rmdir /etc 2> f4
$ rmdir /etc &> f5
$ rmdir /etc 2> /dev/null
```

Question 2. Quelle est la différence entre ces 5 commandes ? Testez aussi en remplaçant **rmdir /etc** par **ls /etc** dans chaque commande.

Testez les commandes suivantes :

```
$ cat > f6 < f1
$ cat > f7 << EOF
Ici écrire
un texte sur plusieurs lignes
et qui se termine par
EOF
$ cat > f8 <<< "Bonjour, comment allez-vous?"
```

Question 3. Quelle est la différence entre `<`, `<<` et `<<<` ?

3 Pipes

Le symbole `|` permet de connecter la sortie standard d'une commande (à gauche du pipe) à l'entrée standard d'une seconde commande (à droite du pipe).

Les pipes sont très utiles pour combiner plusieurs commandes.

Par exemple :

```
$ cat f2 | wc -l
```

Question 1. Que fait la commande ci-dessus ? Pensez à **man wc**.

Autre exemple :

```
$ ls /bin | grep cp | head -n 1
```

Question 2. Que fait la commande ci-dessus ? Pour comprendre, testez aussi `ls /bin` et `ls /bin | grep cp`.

4 Écriture de scripts bash

Un script bash est un fichier contenant une ou plusieurs commandes bash.

Chaque script bash doit commencer par le Shebang (symbole `#!`), qui permet de signaler l'emplacement de l'interpréteur de commande à utiliser pour ce script. Pour un script bash, ce sera `/bin/bash`.

Exemple de création d'un script simple :

```
$ cat > mon_script.sh << EOF
#!/bin/bash
mkdir tmpdir
mv f* tmpdir
EOF
```

Le script peut être exécuté comme suit :

```
$ bash mon_script.sh
```

Question 1. Que fait ce script ?

Bien sûr, il est peu pratique d'écrire des scripts directement dans le terminal. Vous pouvez utiliser l'éditeur de texte de votre choix, par exemple **gedit**, pour écrire vos scripts.

Voici un autre exemple de script :

```
$ cat > mon_autre_script.sh << EOF
#!/bin/bash
ls $1 > $2
EOF
```

Testez ce script en exécutant, par exemple, `bash mon_autre_script.sh /etc resultat`.

Question 2. Que fait ce script ?

Un dernier exemple, plus difficile :

```
$ cat > mon_troisieme_script.sh << EOF
#!/bin/bash
echo $1
if [[ $1 -gt 0 ]]; then
    let n=$1-1
    bash $0 $n
fi
EOF
```

Testez avec `bash mon_troisieme_script.sh 5`.

Question 3. Que fait ce script ?

5 Découpage de chaîne avec cut, head et tail

Écrire un script **get_mask.sh** qui affiche le masque de sous-réseau d'une interface réseau fournie en paramètre.

Exemple :

```
$ bash get_mask.sh eth0
24
```

Vous utiliserez uniquement les commandes **cut**, **head**, **tail** et **ip**. Pour une solution plus propre vous pouvez égalements utiliser la commande **xargs**, qui permet de supprimer les espaces en trop au début d'une chaîne.

La solution peut tenir sur une seule ligne, à l'aide de pipes.

Pensez à utiliser **man** pour trouver les options utiles avec chaque commande.

6 Tri avec la commande sort

Écrire un script **proc_mem.sh** qui affiche la commande utilisant actuellement le plus de mémoire vive.

Exemple :

```
$ bash proc_mem.sh
/usr/bin/mariadb
$ firefox &
$ bash proc_mem.sh
/usr/lib/firefox/firefox
```

Le symbole `&` après **firefox** permet de lancer la commande en tâche de fond, sans monopoliser le terminal courant.

Comme à l'exercice précédent, vous pouvez vous appuyer sur **cut**, **head** et **tail**, mais aussi **sort**. La commande **ps aux** vous fournira toutes les informations sur les processus en cours d'exécution et la mémoire qu'ils occupent.

Vous pouvez aussi utiliser **tr -s ' '**, qui permet de remplacer les groupes d'espaces adjacents par un seul espace.

Attention : le tri numérique effectué par **sort** s'appuie sur votre locale système pour déterminer le symbole qui sépare la partie entière de la partie décimale d'un nombre flottant. En français, c'est donc la virgule, mais la sortie de **ps** utilise des points. Vous pouvez régler ce problème en définissant la variable d'environnement **LC_ALL=C** avant d'appeler **sort**.

7 Permissions Unix et chmod

Pour voir les permissions associées à un fichier, il faut utiliser **ls** avec l'option **-l**.

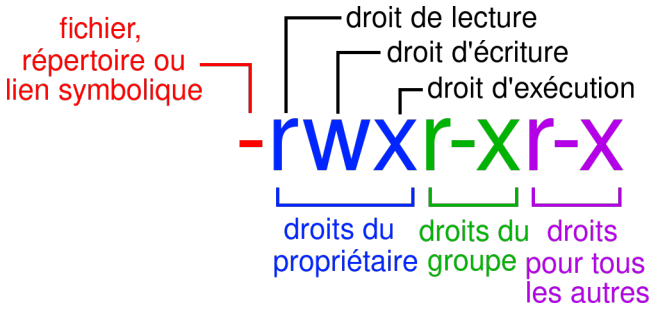
```
$ ls -l
-rwxr-xr-x 1 elise elise 177 18 déc.  14:17 get_mask.sh
-rwxr-xr-x 1 elise elise 164 18 déc.  15:29 proc_mem.sh
```

La première colonne affichée donne les permissions du fichier (exemple ici : `-rwxr-xr-x`).

Le premier caractère indique le type de fichier : la lettre `d` indique un répertoire, `l` indique un lien symbolique, et `-` indique un fichier.

Les 9 autres caractères indiquent les permissions du fichier. Les 3 premiers caractères donnent les permissions pour le propriétaire du fichier, les 3 suivants donnent les permissions pour le groupe propriétaire du fichier, et les 3 derniers caractères donnent les permissions pour tous les autres utilisateurs. Un `r` indique le droit de lecture, un `w` indique le droit d'écriture, et un `x` indique le droit d'exécution. Un `-` indique l'absence du droit correspondant.

Les droits d'exécution permettent d'exécuter un script sans appeler la commande bash, en utilisant par exemple **./proc_mem.sh** au lieu de **bash proc_mem.sh**.



Dans cet exemple, tout le monde a le droit de lire et d'exécuter le fichier, mais seul le propriétaire peut le modifier.

Les permissions d'un fichier peuvent être modifiées à l'aide de la commande **chmod**. Il y a deux façons d'utiliser **chmod** : le mode numérique, qui écrase les permissions existantes, et le mode symbolique, qui permet de modifier une ou plusieurs permissions sur le fichier. Voici quelques exemples.

```
$ touch test
$ ls -l test
-rw-r--r-- 1 elise elise 0 25 déc.  15:00 test
$ chmod 660 test
$ ls -l test
-rw-rw---- 1 elise elise 0 25 déc.  15:00 test
$ chmod +x test
$ ls -l test
-rwxrwx--x 1 elise elise 0 25 déc.  15:00 test
$ chmod g-w test
$ ls -l test
-rwxr-x--x 1 elise elise 0 25 déc.  15:00 test
```

Avec le mode numérique, on peut fournir 3 chiffres à **chmod** (ici `660`). Le premier chiffre indique les nouvelles permissions pour le propriétaire, le second chiffre donne les permissions pour le groupe et le troisième chiffre donne les permissions pour les autres utilisateurs. Chaque chiffre est la somme des droits accordés, en comptant `4` pour le droit de lecture, `2` pour le droit d'écriture, et `1` pour le droit d'exécution. Dans l'exemple, `660` indique que l'on souhaite donner les droits de lecture et d'écriture (`6=4+2`) au propriétaire et au groupe, et aucun droit aux autres utilisateurs.

Avec le mode symbolique, l'opérateur `+` ou `-` indique si l'on souhaite ajouter ou supprimer une permission. Les caractères présents à gauche de l'opérateur indique qui va voir ses droits modifiés `u` pour le propriétaire, `g` pour le groupe, `o` pour les autres utilisateurs, et rien pour tout le monde. Les caractères à droite de l'opérateur indiquent quels droits sont ajoutés ou supprimés (`r`, `w` ou `x`).

Consultez **man chmod** pour plus de détails.

8 Superutilisateur

La plupart des fichiers systèmes (en dehors du dossier `/home/`) sont la propriété de l'utilisateur `root`. Cet utilisateur a tous les droits sur la machine.

De nombreuses tâches d'administration nécessitent d'être connecté sous l'utilisateur `root`, afin de pouvoir modifier les fichiers systèmes. La commande **su** permet de se connecter sous `root`, après avoir tapé le mot passe de l'utilisateur `root`. Une fois connecté sous `root`, le prompt change de `$` en `#`. Vous pouvez vous déconnecter de `root` en tapant la commande **exit**. Exemple :

```
$ touch /fichier
touch: impossible de faire un touch '/fichier': Permission non accordée
$ su
# touch /fichier
# exit
$ ls / | grep fichier
fichier
```

Si vous avez besoin de lancer une seule commande avec les droits `root`, vous pouvez le faire sans changer d'utilisateur grâce à la commande **sudo**. Exemple :

```
$ touch /fichier2
touch: impossible de faire un touch '/fichier2': Permission non accordée
$ sudo touch /fichier2
$ ls / | grep fichier2
fichier2
```

9 Utilisateurs et processus

Utilisez la commande **useradd** pour créer un nouveau compte utilisateur (appellons-le `bob`), puis la commande **passwd** pour attribuer un mot de passe à ce nouvel utilisateur. Attention, les deux commandes nécessitent les droits administrateurs : il vous faudra donc utiliser **sudo** ou bien **su**. Vérifiez que l'utilisateur a été crée en consultant le fichier `/etc/passwd`.

L'utilisateur `bob` aura besoin d'un dossier personnel : créez le dossier `/home/bob` (commande **mkdir**) puis faites de `bob` son propriétaire (commande **chown**). Vous aurez à nouveau besoin des droits administrateurs.

Une fois le nouvel utilisateur créé, vous pouvez utiliser **su bob** pour changer d'utilisateur dans le terminal courant. Lancez ensuite la commande **top**.

Lancez un nouveau terminal (qui n'est donc pas connecté sous `bob`), puis utilisez **ps** pour consulter les processus en cours d'exécution.

Question 1. Quelle différence constatez-vous entre **ps**, **ps -u**, **ps -au** et **ps -aux** ?

Dans le terminal de `bob`, quittez **top** (touche `q`) puis créez un nouveau fichier (commande **touch**). Vous devrez créer ce fichier dans `/home/bob/`, puisque `bob` n'a pas le droit de créer des fichiers ailleurs.

Question 2. En quoi ce nouveau fichier est-il différent des autres ?

Consultez le fichier `/etc/group`.

Question 3. Votre utilisateur courant et `bob` appartiennent-ils au même groupe ?

Si non, ajoutez-les à un groupe commun (par exemple `users`) à l'aide de la commande **usermod**.

Question 4. L'utilisateur courant peut-il désormais modifier le fichier crée par `bob` ? Pourquoi ?

Question 5. Est-il possible qu'un fichier appartienne à un groupe auquel son propriétaire n'appartient pas ? Testez avez **chown**.

10 Permissions spéciales

En plus des permissions classiques vues précédemment, il existe trois permissions spéciales qui peuvent être appliquées à un fichier : le SUID (Set User ID), le SGID (Set Group ID), et le Sticky Bit. Le SUID est indiqué

par un `s` à la place du droit d'exécution pour le propriétaire du fichier, et le SGID est également représenté par un `s`, mais pour le droit d'exécution du groupe. Finalement, le Sticky Bit est représenté par un `t` à la place du droit d'exécution pour les autres utilisateurs. Exemple :

```
$ touch tmp
$ chmod +x tmp
$ ls -l tmp
-rwxr-xr-x 1 elise elise 0 3 janv. 14:45 tmp
$ chmod u+s tmp
$ ls -l tmp
-rwsr-xr-x 1 elise elise 0 3 janv. 14:45 tmp
$ chmod g+x tmp
$ ls -l tmp
-rwsr-sr-x 1 elise elise 0 3 janv. 14:45 tmp
$ chmod o+t tmp
$ ls -l tmp
-rwsr-sr-t 1 elise elise 0 3 janv. 14:45 tmp
```

Chacun des trois types de permissions spéciales a un effet différent. Testons tout d'abord le SUID. Pour des raisons de sécurité, le SUID n'a pas d'effet sur les scripts bash : nous allons donc créer un petit programme C.

```
$ cat > wait.c << EOF
#include <unistd.h>
int main() {
    while (1)
        sleep(1);
    return 0;
}
EOF
$ gcc -o wait wait.c
```

Il s'agit d'une simple boucle infinie. N'oubliez pas de compiler le programme. Nous allons en faire deux copies : l'une aura le droit SUID, et l'autre non. L'utilisateur `bob` va se charger d'exécuter les deux programmes.

```
$ cp wait wait2
$ chmod u+s wait2
$ su bob
$ ./wait &
$ ./wait2 &
$ ps aux | grep wait
```

Question 1. Que constatez vous en lisant la sortie du **ps aux** ? D'après vous, quel est l'effet du SUID ?

Vous pourrez interrompre les programmes qui tournent en tâche de fond avec **killall wait** et **killall wait2**.

Le SGID peut être utile pour travailler en groupe dans un même dossier. En effet, si un dossier dispose du droit SGID, alors tout fichier crée dans ce dossier appartiendra au groupe propriétaire du dossier. Exemple :

```
$ mkdir dir
$ chmod 777 dir
$ chmod g+s dir
$ su bob
$ touch dir/test
$ ls -l dir
total 0
-rw-r--r-- 1 bob elise 0 3 janv. 15:19 test
```

Finalement, le Sticky Bit s'applique également sur des répertoires. Dans un système Linux, le dossier `/tmp` dispose du Sticky Bit. Testez les commandes suivantes :

```
$ touch /tmp/teststicky
$ su bob
$ rm /tmp/teststicky
$ touch /tmp/teststickybob
$ exit
$ rm /tmp/teststicky
```

Question 2. Que concluez vous sur l'effet du Sticky Bit sur un répertoire ?

11 Variables et environnement

Les variables d'environnement sont des variables utilisées par le système et qui sont accessibles aux scripts bash. Vous pouvez voir toutes les variables d'environnement actuellement existantes avec la commande **printenv**. Testez cette commande pour voir le résultat.

Vous pouvez aussi afficher la valeur d'une variable d'environnement en particulier comme suit :

```
$ echo $USER
elise
```

Les variables en bash ont une portée limitée. Elles sont, par défaut, locales au script qui les référence. Cependant ce comportement peut être modifié grâce aux mots-clés `export` et `source`. Par exemple :

vars1.sh

```
#!/bin/bash
n1=5
n2=10
n3=11
export n1 n3
echo "Sans source"
bash vars2.sh
echo "vars1 n1=$n1 n2=$n2 n3=$n3"
echo "Avec source"
source vars2.sh
echo "vars1 n1=$n1 n2=$n2 n3=$n3"
```

vars2.sh

```
#!/bin/bash
n3=12
echo "vars2 n1=$n1 n2=$n2 n3=$n3"
```

Question 1. Exécutez le script **vars1.sh**. En étudiant le résultat, que pouvez vous déduire du fonctionnement de `export` et `source` ?

12 Calculs sur des variables

Bash fournit des outils très pratiques pour manipuler le contenu des variables chaînes de caractères.

Exécutez le script suivant et essayez de comprendre ce que fait chaque ligne :

```
#!/bin/bash
var="chemin/vers/un/fichier/imaginaire"
echo $var
echo ${#var}
echo ${var#chemin/}
echo ${var#*/}
echo ${var##*/}
echo ${var%/imaginaire}
echo ${var%%/*}
echo ${var:-"au cas où"}
echo ${truc:-"au cas où"}
echo ${var:3:5}
echo ${var/fichier/dossier}
```

13 PATH

Lorsqu'une commande est entrée dans le terminal, bash consulte la variable d'environnement $\$PATH$ pour trouver le fichier exécutable correspondant à la commande. Testez **echo** $\$PATH$ pour voir la liste des emplacements où les commandes exécutables sont stockées.

On souhaite faire de **proc_mem.sh** une commande bash. Pour cela nous allons créer un nouveau dossier *scripts* auquel nous ajouterons nos nouvelles commandes bash.

```
$ mkdir ~/scripts/
$ mv proc_mem.sh ~/scripts/proc_mem
$ chmod +x ~/scripts/proc_mem
```

Il ne reste plus qu'à modifier la variable $PATH$ pour qu'elle pointe vers le dossier ~/scripts/.

Attention, modifier $PATH$ peut casser le système ! En effet, $PATH$ est utilisée non seulement pour vos commandes personnalisées mais aussi pour les commandes de base du système telles que **cd** ou **ls**. Si $PATH$ ne contient plus l'emplacement de ces commandes, le système ne peut plus fonctionner correctement. Lorsque vous modifiez $PATH$, prenez soin de ne pas supprimer les emplacements existants.

Ajoutez le dossier ~/scripts/ à $PATH$ et vérifiez que la commande **proc_mem** fonctionne.

La modification de $PATH$ n'est valable que dans le terminal courant. Pour la rendre permanente, modifiez $PATH$ dans ~/.bashrc : la modification sera alors faite à chaque ouverture d'un nouveau terminal. Assurez-vous d'abord de ne pas avoir cassé le système... Est-ce que **ls** fonctionne toujours ?

Créez un dossier ~/scripts2/ puis ajoutez-le également à $PATH$ et copiez **proc_mem** dans ce nouveau dossier. Lorsque vous exécutez **proc_mem**, quelle version du script est exécutée ? Vous pouvez utiliser la commande **which** pour vérifier.

Question 1. Comment bash choisit-il la version du script à utiliser ?

14 Alias

La commande **alias** est une autre façon de créer de nouvelles commandes, ou de modifier les commandes existantes. Vous pouvez l'utiliser comme suit :

```
$ macommande
bash: macommande : commande introuvable
$ alias macommande="pwd"
$ macommande
/home/elise
```

Question 1. Testez les commandes ci-dessus, puis fermez le terminal courant. Ouvrez un nouveau terminal, puis exécutez **macommande**. Que se passe-t-il ?

Le fichier **.bashrc** situé dans votre dossier personnel est un script qui est exécuté à chaque lancement d'un nouveau terminal. Éditez ce fichier (ou créez le s'il n'existe pas) et ajoutez-y la commande suivante :

```
alias ls='ls --color=auto'
```

Cette modification imposera à la commande **ls** de toujours inclure l'option **--color=auto**, qui ajoute quelques couleurs à l'affichage de **ls**.

N'hésitez pas à ajouter d'autres alias dans votre **.bashrc**.

15 Codes de retour

Un script bash peut retourner une valeur à l'aide du mot-clé **exit**. Cette valeur peut ensuite être récupérée grâce à la variable `$?`. Exemple :

```
$ cat > retour.sh << EOF
#!/bin/bash
exit 3
EOF
$ bash retour.sh
$ echo $?
3
```

Question 1. Quel est le code de retour de chacune des commandes suivantes ?

```
$ ls | grep "proc_mem.sh"
$ ls | grep "machin"
$ rm machin
$ touch machin
```

Question 2. D'après vous, quelles valeurs numériques sont utilisées en bash pour signifier "vrai" et "faux" ?

Question 3. Si un script n'appelle jamais la commande **exit**, quelle valeur retourne-t-il ?

16 Conditionnelles et comparaisons

En bash le mot-clé **if** suivi d'une commande permet de tester le code de retour de cette commande et d'exécuter le code qui suit conditionnellement suivant la valeur trouvée. Par exemple :

```
if ls | grep "$1" > /dev/null; then
    exit 0
else
    exit 1
fi
```

Question 1. Que fait le script ci-dessus ?

if fonctionne également également avec `[[` pour vérifier des conditions simples. Par exemple :

```
if [[ $var -eq 2 ]]; then
    echo "$var est égal à 2"
fi
```

`[[` supporte les opérateurs de comparaison **-eq** (égal à), **-ne** (différent de), **-gt** (plus grand que), **-ge** (plus grand ou égal), **-lt** (plus petit que) et **-le** (plus petit ou égal).

Vous pouvez aussi utiliser **-f**, qui vérifie si l'emplacement fourni existe et est un fichier, et **-d**, qui vérifie si l'emplacement fourni existe et est un dossier. Par exemple :

```
if [[ -f $1 ]]; then
    echo "Le fichier existe!"
fi
```

Dans certains cas, on peut se passer complètement de **if** grâce aux symboles `&&` et `||`. Le symbole `&&` permet d'exécuter une seconde commande seulement si une première commande a retourné 0, et `||` permet d'exécuter une seconde commande seulement si la première a retourné autre chose que 0. Exemple :

```
$ ls | grep "proc_mem.sh" > /dev/null && echo "Trouvé!"
Trouvé!
$ [[ 1 -eq 0 ]] || echo "Raté."
Raté.
```

Question 2. Écrivez un script qui prend un argument numérique, et affiche "Valide" si l'argument est compris entre 0 et 10, et "Invalide" autrement.

17 Différents types de guillemets en bash

En bash, il existe trois types de guillemets différents : les simple quotes (`'`), les double quotes (`"`) et les backquotes (`` ` ``, obtenu avec les touches **AltGr+7**).

Voici deux scripts illustrant la différence entre ces trois types de guillemets :

hello.sh

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

guillemets.sh

```
#!/bin/bash
var="bash hello.sh"
echo '$var' # simples
echo "$var" # doubles
echo `$var` # backquotes
```

Question 1. Quel affichage produit l'exécution de **guillemets.sh** ? Pourquoi ?

Question 2. Écrivez un script **delegation.sh** qui affiche un message, puis affiche le résultat d'une commande écrite dans un fichier fourni en argument.

Exemple d'exécution :

```
$ echo "./get_mask.sh eth0" > commande
$ ./delegation.sh commande
Test: 24
$ echo "./proc_mem.sh" > commande
$ ./delegation.sh commande
Test: /usr/lib/firefox/firefox
```

Indice : imbriquer des backquotes dans des backquotes n'aura pas d'effet. En revanche, vous pouvez utiliser des backquotes pour appeler une commande et stocker le résultat dans une variable, puis utiliser à nouveau des backquotes autour de cette variable.

18 Date

La commande **date** permet d'afficher la date et l'heure. Les arguments passés à la commande permettent de formater l'affichage. Lisez la section Format dans **man date** pour plus de détails.

Écrivez un script **format_date.sh** qui utilise la commande **date** pour produire l'affichage suivant :

```
$ ./format_date.sh
mardi 3 janvier 2023, 16:13:02 (UTC+0100)
La date actuelle est 03-01-2023
Il est actuellement 16:13:02
```

19 Boucles

La syntaxe des boucles **while** et **for** ne présente pas de surprise.

```
for ((i=0; i<10; i++)) do
    echo $i
done

i=0
while [[ $i -lt 10 ]]; do
    echo $i
    let i=$i+1
done
```

On peut également utiliser une autre syntaxe pour les boucles **for** :

```
for i in 1 2 3 4 5 6 7 8 9 10; do
    echo $i
done
```

C'est particulièrement utile lorsqu'on utilise des backquotes :

```
for file in `ls`; do
    echo "Test: $file"
done
```

Question 1. Testez la boucle ci-dessus : qu'affiche-t-elle ?

Question 2. Écrivez un script qui copie tous les fichiers du répertoire courant, en ajoutant un numéro incrémental à la fin du nom de chaque fichier (par exemple, si **delegation.sh** est le 4è fichier trouvé, il est copié en **delegation.sh4**).

20 Inversion

Écrivez un script **inverse.sh** qui inverse les lignes d'un fichier et stocke le résultat dans un second fichier. Par exemple :

```
$ cat fichier1
Ce fichier
contient plusieurs
lignes de texte
$ ./inverse.sh fichier1 fichier2
$ cat fichier2
lignes de texte
contient plusieurs
Ce fichier
```

Vous pourrez utiliser la commande **read**, qui permet de parcourir un fichier ligne par ligne comme dans l'exemple suivant :

```
while read line; do
    echo "$line"
done < fichier_a_lire
```

La boucle ci-dessus est équivalent à **cat fichier_a_lire**.

21 Factorielle

Écrivez un script qui affiche la factorielle d'un nombre fourni en argument. Programmez d'abord ce script en version itérative (à l'aide d'une boucle for) puis en version récursive (à l'aide de backquotes).

Vous aurez besoin de faire des calculs arithmétiques. Pour cela, vous pouvez utiliser **let** comme dans l'exemple suivant :

```
$ let n1=3+2
$ let n2=$n1*4
$ echo $n2
20
```

Consultez **let --help** pour plus d'informations.

22 Find

Question 1. Testez la commande **find** suivante. Que fait elle ?

```
$ find /bin/ -name "*cp*"
```

Question 2. Même question avec la commande suivante.

```
$ mkdir tmpdir
$ find . -name "g*" -exec cp {} tmpdir \;
```

Question 3. Écrivez une commande **find** qui exécute chaque script (fichier .sh) du répertoire courant et redirige le résultat dans un fichier nommé `out`.

A Annexe : commandes de bases

man `commande`	Affiche le manuel de la commande `commande`.
pwd	Affiche l'emplacement du dossier courant.
ls	Affiche le contenu du dossier courant. Options utiles : **-l**, **-h**, **-a**...
ls `dossier`	Affiche le contenu de `dossier`.
cd `dossier`	"Change Directory" : change le dossier courant, qui devient `dossier`.
cat `fichier`	Affiche le contenu de `fichier`.
mkdir `dossier`	Crée le dossier `dossier`.
mv `f1 f2`	Déplace (ou renomme, c'est la même chose) le fichier `f1` en `f2`.
cp `f1 f2`	Copie le fichier `f1`. La copie s'appellera `f2`.
cp -r `d1 d2`	Copie récursivement le dossier `d1` et tout son contenu. La copie s'appellera `d2`.
rm `fichier`	Supprime le fichier `fichier`.
rm -r `dossier`	Supprimer récursivement le dossier `dossier` et tout son contenu.
rmdir `dossier`	Supprime le dossier `dossier`, mais seulement si il est vide.
touch `fichier`	Met à jour la date de dernière modification de `fichier`. Si le fichier n'existe pas, il est crée.
echo `"Bonjour!"`	Affiche `Bonjour!` sur la sortie standard.
head -n 3	Affiche seulement les 3 premières lignes du texte fourni en entrée standard.
tail -n 3	Affiche seulement les 3 dernières lignes du texte fourni en entrée standard.
cut -d',' -f2	Découpe le texte fourni en entrée standard en colonnes (en utilisant les virgules comme séparateurs), et affiche seulement la seconde colonne.