

Introduction aux systèmes d'exploitation (IS1)

TP n° 11 : scripts

Le but de ce TP est d'écrire de petits programmes utilitaires – appelés couramment *scripts* – en bash, un peu plus complexes que ceux des TP précédents; bash est en effet plus qu'un langage permettant d'exécuter des commandes, et possède les caractéristiques d'un langage de programmation (rudimentaire, certes) : on peut définir des variables, des fonctions, écrire des instructions conditionnelles, des boucles... Pour une documentation complète, voir [la documentation en ligne de bash](#).

Un script contient des séquences de commandes telles que l'on pourrait les taper dans un terminal. Les commandes successives sont séparées par des retours à la ligne.

Un fichier de script commence normalement par une ligne permettant d'identifier le programme qui doit l'interpréter – dans le cas d'un script bash, une ligne équivalente à `#!/usr/bin/env bash`. En l'absence de telle ligne, le script est interprété par un shell par défaut (souvent sh, bash ou dash), ce qui ne convient pas toujours (pour un script en python, en perl, ou utilisant des particularités d'un shell spécifique plus évolué). Il est donc souhaitable de prendre l'habitude d'écrire systématiquement cette ligne.

Pour qu'un utilisateur puisse exécuter un script, il doit posséder les droits en **exécution**, mais aussi en **lecture** sur ce script (c'est un cas particulier où la lecture est nécessaire à l'exécution).

Modalités de rendu Déposer sur Moodle une archive `archive_TP11.tar` contenant les réponses aux questions marquées par le symbole 🚩 dans un fichier `reponses_TP11.txt`, ainsi que **tous les scripts** demandés.

Installation préalable nous allons utiliser la commande « convert » qui permet de faire des manipulations sur des images. *Si* elle n'est pas encore présente sur votre machine, il faut installer le paquet [ImageMagick](#), ce qui peut se faire par exemple par :

- « `sudo apt update` » puis « `sudo apt install imagemagick` » sous ubuntu,
- « `brew update` » puis « `brew install imagemagick` » sous macOS,

et de manière analogue si vous utilisez un autre gestionnaire de paquets.

Si c'est un peu long, laissez l'installation se faire tranquillement et passez à la suite depuis un autre terminal.

Terminaison et valeur de retour

Tout processus UNIX renvoie à son processus père une *valeur de retour* indiquant les conditions de son arrêt. Cette valeur est un entier positif, par convention égal à 0 **si et seulement si** l'exécution et la terminaison se sont déroulées correctement. Les autres valeurs de retour possibles d'une commande sont documentées dans son manuel.

La variable d'environnement « ? » contient la valeur de retour de la précédente commande exécutée par le shell.

Exercice 1 – « echo \$? »

1. Exécuter « ls ». Quelle est la valeur de retour du processus ? Recommencer avec un nom de fichier qui n'existe pas, par exemple « ls grosminet ». ➤ Comparer.
2. Exécuter « grep a », taper une ligne contenant un a, et terminer proprement (par *ctrl-D*). Quelle est la valeur de retour du processus ? Recommencer en saisissant cette fois des lignes sans a, puis avec « echo aaa | grep a » et « echo bbb | grep a ». ➤ De quel processus obtenez-vous la valeur de retour dans ces deux derniers cas ?
3. Exécuter à nouveau « grep a », et interrompre le processus à l'aide de *ctrl-C*. ➤ Quelle est sa valeur de retour ? Recommencer en interrompant le processus à l'aide de différents signaux.

Les enchaînements conditionnés

Vous avez peut-être déjà remarqué que « cmd1 ; cmd2 » exécute « cmd2 » une fois que « cmd1 » a terminé. Deux autres connecteurs permettent d'enchaîner deux commandes, mais en conditionnant l'exécution de la deuxième à la valeur de retour de la première :

commande₁ && commande₂ && ... && commande_n

Le shell exécute alors *commande₁*, puis, une fois celle-ci terminée, si sa valeur de retour est nulle, il exécute *commande₂*, et ainsi de suite jusqu'à la dernière commande de la liste. Cette méthode permet d'enchaîner les commandes ***tant que*** tout se déroule correctement.

commande₁ || commande₂ || ... || commande_n

Le shell exécute *commande₁*, puis, une fois celle-ci terminée, si sa valeur de retour est non nulle, il exécute *commande₂*, et ainsi de suite jusqu'à la dernière commande de la liste. Cette méthode permet d'enchaîner les commandes ***jusqu'à ce qu'***une d'entre elles s'exécute sans erreur.

Exercice 2 – enchaînements conditionnés

1. Exécuter « sleep 5 || xclock », puis « sleep 5 && xclock », sans interrompre la commande « sleep ». ➤ Comparer, puis tuer le(s) processus qui reste(nt).
2. Exécuter ensuite « sleep 500 || xclock », puis « sleep 500 && xclock », en interrompant la commande « sleep » par l'envoi d'un signal. ➤ Comparer les réactions, puis tuer le(s) processus qui reste(nt).

Exercice 3 – enchaînements et valeur de retour

1. Exécuter la suite de commandes « `ls grosminet ; echo "pas de chat" »`. Quelle est la valeur de la variable « `?` ». Expliquez.
2. Faire de même avec « `ls grosminet || echo "pas de chat" »` et « `ls grosminet && echo "pas de chat" »` et expliquez.

Exercice 4 – comparer deux fichiers

Dans cet exercice, nous allons utiliser la commande « `diff` » avec l’option `-q`, qui permet de tester si deux fichiers sont identiques.

1. Créer deux fichiers `pareil` et `mime` au contenu identique, et un fichier `different` au contenu différent. ➤ Comparer le comportement et la valeur de retour de la commande « `diff -q` » selon qu’elle a été appelée avec deux fichiers identiques ou différents. Où le message éventuel s’affiche-t-il ?
2. Écrire une séquence d’instructions qui compare deux fichiers et affiche
Les deux fichiers sont identiques.
quand c’est le cas, et rien dans le cas contraire (donc en particulier sans afficher le message associé à « `diff -q` »).
3. ➤ Réciproquement, écrire une commande qui affiche le message
Les deux fichiers sont différents.
(uniquement) quand c’est le cas.
4. ➤ Combiner ces deux séquences pour afficher la phrase correcte en fonction du résultat de la commande, toujours sans afficher le message associé à « `diff` ».

Paramètres d’un script

Comme vous le savez, la plupart des commandes acceptent des paramètres et/ou des options, qui leur sont transmis par le shell après interprétation et découpage de la ligne de commande. Lors de l’écriture d’un programme (en Java, C, bash, ou tout autre langage), il est possible de manipuler ces paramètres (qui seront transmis au programme lors de son exécution). En Java par exemple, cette transmission se fait par le biais du paramètre `String[] args` du `main`. En bash¹, cela se fait via certaines variables spéciales, dites *réservées* car elles ne peuvent être initialisées que par le shell et non au moyen d’une affectation explicite « `var=valeur` » ; en revanche leur valeur est consultable par `$var` comme toute autre variable.

1. comme dans les autres shells, d’ailleurs

Les paramètres

- \$# est le nombre de paramètres passés au script,
- \$0 est le nom de la commande en cours d'exécution,
- pour chaque entier i entre 1 et \$#, \$i contient le i^e paramètre,
- @\$ contient la liste de tous les paramètres.

Il faut en général protéger ces variables avec des guillemets (notamment "\$@"), à cause des espaces éventuels dans les paramètres.

Par exemple, la ligne `mon_script.sh fic1 fic2 fic3` est découpée en une liste de quatre chaînes de caractères `mon_script.sh` `fic1` `fic2` `fic3`, et donc \$#=3, \$0=mon_script.sh, \$1=fic1, \$2=fic2 et \$3=fic3.

Exercice 5 – paramètres d'un script

1. Écrire un script `salut.sh` qui, lorsqu'il reçoit un nom *tartempion* en paramètre, affiche Salut, *tartempion* ! Par exemple, la ligne de commande « `./salut.sh Gontran` » doit afficher Salut, Gontran !
2. Écrire un script `bonjour.sh` qui affiche Je dois saluer ... personnes : (en remplaçant ... par le nombre d'arguments reçus), puis sur la ligne suivante : Bonjour, suivi de tous les noms reçus en paramètre.
Par exemple, « `./bonjour.sh Riri Fifi Loulou` » doit afficher :
Je dois saluer 3 personnes :
Bonjour, Riri Fifi Loulou !

Bien entendu, un traitement satisfaisant des paramètres n'est pas vraiment envisageable sans tests ni boucles ; de telles structures existent fort heureusement en bash.

Les structures de contrôle

Les conditions peuvent être gérées de plusieurs manières en bash, mais nous ne verrons que celle qui se rapproche le plus des autres langages comme Java :

Les conditionnelles La syntaxe des instructions conditionnelles est la suivante :

```
if test
then commande1 (ou 1er bloc de commandes)
else commande2 (ou 2e bloc de commandes)
fi
```

Les retours à la ligne sont obligatoires, la partie `else` est facultative.

Les tests eux-mêmes peuvent prendre de nombreuses formes – en fait, n'importe quelle commande peut servir, car sa valeur de retour peut être interprétée comme un booléen. Nous allons cependant nous contenter de la syntaxe ci-dessous, à ne pas confondre avec une syntaxe similaire utilisant une seule paire de crochets, antérieure et nettement moins agréable.

Les tests Une syntaxe possible pour les tests est `[[expression]]`, où *expression* peut faire appel à tout un panel d'opérateurs unaires ou binaires pour tester si un fichier existe (`[[-e fic]]`), si c'est un répertoire (`[[-d fic]]`), si deux chaînes sont égales (`[[expr1 == expr2]]`) ou différentes (`!=`), si deux conditions sont vraies simultanément (`&&`)...

Attention, les espaces sont obligatoires partout où ils sont possibles : entre les crochets et l'expression, autour des opérateurs...

Pour une liste des différents tests possibles, voir [la documentation en ligne de bash](#).

Exercice 6 – premier test

1. Modifier `bonjour.sh` pour qu'il vérifie qu'il a bien reçu des paramètres. Dans le cas contraire, il devra afficher un message expliquant le bon usage de la commande.

« `exit val` » termine le processus avec la valeur de retour *val*.

2. Modifier `bonjour.sh` pour qu'il termine avec la valeur de retour 42 si aucun paramètre ne lui a été fourni, et 0 sinon.
Vérifier que les processus créés en exécutant ce script ont le comportement attendu.

Exercice 7 – miniaturisation d'images

Le but de cet exercice est d'écrire un script `miniaturise.sh` prenant en paramètre la référence d'un fichier image, et créant une icône à partir de celle-ci.

Pour cela nous allons utiliser la commande « `convert` » (ou « `magick convert` ») et son option « `-thumbnail` » (i.e. icône en anglais). Celle-ci prend un argument *geometry* qui peut être un format (*largeurxhauteur*) ou juste une dimension (*largeur* ou *xhauteur*).

1. Choisir une image *img*, puis à l'aide de « `convert` », créer une icône *mini_img* de largeur fixée (par exemple 100 pixels) à partir de *img*.
2. Écrire un script `miniaturise.sh` qui prend en paramètre la référence d'un fichier image dans le répertoire courant, et crée l'icône correspondante, en ajoutant le préfixe *mini_* au nom de l'image. Le script ne vérifiera pas que le paramètre est correct pour le moment.
3. Modifier le script pour qu'il vérifie qu'il a bien reçu (exactement) un paramètre, et que ce paramètre est une référence valide d'un fichier ordinaire. Dans le cas contraire, un message précisera à l'utilisateur le bon usage du script, puis le script terminera avec la valeur de retour 1.
4. Modifier le script pour qu'il vérifie que la référence passée en paramètre correspond bien à un fichier dans le répertoire courant, grâce à un test sur la sortie de la commande « `dirname` », et affiche un message dans le cas contraire avant de terminer avec la valeur de retour 2.

5. Modifier le script pour qu'il vérifie (toujours avant de créer l'icône !) que le paramètre qui lui est fourni est bien la référence d'un fichier image. Dans le cas contraire, un message indiquera le problème à l'utilisateur, puis le script terminera avec la valeur de retour 3. Pour réaliser ce test, vous pouvez par exemple utiliser une combinaison de la commande « `file` » et de « `grep` », et, au choix, utiliser la valeur de retour de cette combinaison, ou stocker sa sortie dans une variable `tmp` dont vous testerez ensuite la valeur.
6. Modifier le script pour qu'il crée un sous-répertoire Miniatures dans le répertoire courant s'il n'existe pas encore. Si un fichier autre qu'un répertoire porte ce nom, le script doit terminer avec un message d'erreur et la valeur de retour 4, sans créer d'icône. Si tout est en ordre, l'icône doit être placée dans ce répertoire.

Les boucles Il en existe plusieurs types, mais nous nous contenterons des boucles de type « pour tout » définies de la manière suivante :

```
for var in liste_de_valeurs
do commande
done
```

La liste de valeurs peut être explicite, ou obtenue par expansion d'une chaîne comme `*`, ou contenue dans une variable, ou obtenue comme résultat d'une commande (à l'aide de `$(cmd)` ou `'cmd'`).

Exercice 8 – première boucle

Modifier `bonjour.sh` pour qu'il salue chaque personne individuellement, c'est-à-dire qu'il traite un paramètre par ligne et non toute la liste en une fois.

Exercice 9 – lister les fichiers images du répertoire courant... ou de toute une arborescence

1. Écrire un script `liste_types.sh` qui liste les fichiers du répertoire courant à l'aide d'une boucle sur (l'expansion de) `*` (même si vous savez le faire plus simplement).
2. Modifier le script `liste_types.sh` pour qu'il affiche le type de chaque fichier à l'aide de la commande « `file` ».
3. Écrire un script `liste_images.sh` qui affiche uniquement les noms des fichiers images présents dans le répertoire courant.
4. Modifier `liste_images.sh` pour qu'il prenne en paramètre la référence du répertoire où chercher les images.
5. (optionnel) Modifier ensuite le script pour que « `liste_images.sh rep` » liste toutes les images contenues dans l'arborescence de racine `rep`. Pour cela, il faut écrire un script **récuratif**, c'est-à-dire qui s'invoque lui-même (avec un paramètre différent).

Exercice 10 – *créer toutes les miniatures*

1. Écrire un script `cree_miniatures.sh` qui crée dans le sous-répertoire `Miniatures` une icône de chaque image du répertoire courant, en créant ce répertoire si besoin. Ce script pourra naturellement faire appel à `miniaturise.sh`.
2. Modifier `cree_miniatures.sh` pour qu'il prenne en paramètre les références du répertoire où se trouvent les images à traiter et du répertoire où le sous-répertoire `Miniatures` doit être créé.

Exercice 11 – *créer un index*

Récupérer sur Moodle le script `cree_index.sh`. Ce script prend les mêmes paramètres que votre script `cree_miniatures.sh`, exécute votre script avec ces paramètres, puis crée dans le répertoire courant un fichier d'index au format html, `miniatures.html`.

Exécuter ce script et ouvrez ensuite `miniatures.html` sur votre navigateur.