Introduction aux systèmes d'exploitation (IS1)

TP 6– Enchaînements de commandes

Exercice 1

Le shell `bash` est un véritable langage de programmation (ou plus précisément un langage de scripts), possédant ses variables, structures de contrôle (conditionnelles, boucles, cas), fonctions, etc. Au fur et à mesure des séances, nous découvrirons certains de ces aspects.

La construction la plus simple dans tout langage de programmation est l'enchaînement séquentiel des commandes. En `bash`, il existe plusieurs possibilités pour lancer plusieurs commandes l'une après l'autre :

- commande$_1$; commande$_2$ ... ; commande$_n$
- commande$_1$ && commande$_2$ ...&& commande$_n$
- commande$_1$ || commande$_2$ ...|| commande$_n$

Nous allons découvrir ici à quoi correspondent ces façons d'enchaîner des commandes, pour cela nous allons utiliser la commande `sleep`, qui lance un processus qui se contente d'attendre le temps qu'on lui indique. Par exemple `sleep 60` attend une minute et `sleep 2` attend deux secondes (voir le manuel pour plus de détails).

1. Exécutez :

```
sleep 2; echo "haha"
```

Que se passe-t-il ? Même question en remplaçant ; par `&&`, puis par `||`.

Correction. Avec ; et `&&`, il se passe 2 secondes puis le terminal affiche haha. Avec `||`, haha ne s'affiche pas.

2. Exécutez :

```
(sleep 60; xcalc) &
```

Quel est le nom de la commande correspondant au PID qui s'affiche ? Quels sont les processus qui viennent d'apparaître en tout ? (utilisez `ps`). Avant la fin de la minute, tuez le processus qui correspond au PID qui s'est affiché. Est-ce que `xcalc` se lance à la fin de la minute ?

Correction. bash. bash et sleep. non.

3. Exécutez à nouveau :

```
(sleep 60; xcalc) &
```

Tuez le processus correspondant à `sleep`. Que se passe-t-il ? Même question en remplaçant ; par `&&`, puis par `||`.

Correction. Avec ;, xcalc se lance. Avec && non. Avec || oui.

4. Le shell peut donc se comporter différemment suivant le succès ou l'échec d'une commande. Pouvez-vous décrire avec des mots comment le shell exécute :

```
commande1; commande2
commande1 && commande2
commande1 || commande2
```

en fonction du succès ou de l'échec de `commande1` ?

Correction. Une suite de commandes séparées par un point virgule est exécutée par le shell de la manière suivante : Le shell exécute la commande 1. Une fois que celle ci est terminée, et quelle que soit sa valeur de retour, il exécute la commande 2.
Une suite de commandes séparées par && ("et") est exécutée par le shell de la manière suivante : Le shell exécute la commande 1. Une fois que celle ci est terminée, et si la valeur de retour est nulle (exécution normale), il exécute la commande 2.
Une suite de commandes séparées par || ("ou") est exécutée par le shell de la manière suivante : Le shell exécute la commande 1. Une fois que celle ci est terminée, et si la valeur de retour est non nulle (exécution spéciale), il exécute la commande 2.

Valeurs de retour

Exercice 2

Dans l'architecture Unix, un processus qui se termine communique en principe à son environnement, et en particulier à son processus père, une information sur les conditions de son arrêt, que l'on appelle valeur de retour. L'exercice précédent est en particulier un exemple où le shell se comporte différemment suivant la valeur de retour des processus qu'il exécute.

Les valeurs de retour possibles pour les différentes commandes sont en général documentées dans la page de manuel correspondante.

Par convention, une valeur de retour nulle (égale à 0) signifie que l'exécution et la terminaison du processus se sont déroulées normalement. Une valeur strictement positive correspond à un cas particulier ou à une erreur.

1. Dans un terminal, la commande :

```
echo $?
```

permet d'afficher le code de retour de la dernière commande exécutée[1]. Quelle est la valeur de retour de `sleep 2` après avoir attendu de récupérer la main ?

Correction. Valeur de retour nulle (exécution normale).

2. Exécutez `sleep 600` et interrompez l'attente avec Ctrl-C. Que dit `echo $?` ? (Tapez à nouveau `echo $?`. Quelle est la nouvelle valeur ? Pourquoi ?)

Correction. Valeur de retour non nulle (exécution interrompue par Ctrl-C). Si on refait `echo $?` on a cette fois une valeur de retour nulle qui correspond à la dernière commande exéctuée, à savoir le précédent `echo $?` tapé, qui s'est déroulé normalement.

3. En remplaçant `commande1` et `commande2` au choix par `sleep 2` et/ou `sleep 600` dans ce qui suit, et en utilisant au besoin Ctrl-C pour faire échouer un processus, donnez un tableau décrivant le succès (déroulement normal) ou l'échec pour chacune des commandes suivantes, en fonction du succès et de l'échec de `commande1` et `commande2` :

```
commande1; commande2
commande1 && commande2
commande1 || commande2
```

Peut-on comparer ces opérateurs à la logique booléenne ?

[1] Cette notation sera expliquée en détail dans le TP sur les variables du shell.

Correction. Cette question est un peu ouverte ; on peut faire le rapprochement avec les tableaux de vérité du "et" et du "ou" lorsque 0 signifie vrai. On peut aussi que remarquer que les commandes ne sont pas toujours exécutées, d'où une asymétrie dans la définition de ces opérateurs, ce qui les distancie de la logique.

4. En déduire une description en quelques mots du comportement de chacune des listes de commandes suivantes :
   commande_1 ; commande_2 ; ... ; commande_n
   commande_1 && commande_2 && ...&& commande_n
   commande_1 || commande_2 || ...|| commande_n

   Correction. ; permet d'enchaîner des commandes. && permet d'enchainer les commandes tant que tout se déroule correctement. || permet d'enchaîner les commandes jusqu'à ce qu'une d'entre elles s'exécute sans erreur.

Exercice 3 – Enchaînements imposés.

1. La commande `cmp` permet de tester si deux fichiers sont identiques. Créez deux fichiers identiques `pareil` et `meme` et un fichier différent `differe`. Comparez le comportement et la valeur de retour de la commande `cmp` selon qu'elle a été appelée avec deux fichiers identiques ou avec deux fichiers différents.

   Correction. Lorsque les fichiers sont identiques, rien ne s'affiche et la valeur de retour est nulle. Lorsque les fichiers sont différents, un message s'affiche et la valeur de retour est non nulle.

2. Le message affiché par la commande `cmp` lorsque les deux fichiers sont différents correspond-il à la sortie standard ou à la sortie erreur standard ?
   (Indice : essayez de rediriger la sortie standard ou à la sortie erreur standard pour le voir.)

   Correction. Cela dépend du système d'exploitation et de votre configuration. Souvent ce message est sur la sortie standard (le message n'est pas un message d'erreur, même si la valeur de retour est non nulle).

3. Ecrivez une séquence d'instructions qui compare deux fichiers de votre répertoire courant et affiche "les deux fichiers sont identiques" quand c'est le cas, et n'affiche rien sinon.

   Correction. `cmp fic1 fic2 && echo "les deux fichiers sont identiques"`

4. Réciproquement, écrivez une commande qui affiche "les deux fichiers sont différents" quand c'est le cas et n'affiche rien sinon.

   Correction. `cmp fic1 fic2 || echo "les deux fichiers sont differents"`

5. Combinez ces deux séquences pour afficher la phrase correcte en fonction du résultat de la commande (indice : vous pouvez délimiter une séquence d'instructions à l'aide de parenthèses). Expliquez pourquoi votre solution fonctionne.

   Correction. `(cmp fic1 fic2 && echo "Ils sont identiques") || echo "Ils sont differents"`

6. Même question en faisant en sorte que seule la phrase demandée s'affiche, et jamais le message associé à la commande `cmp`.

   Correction. `(cmp fic1 fic2 >/dev/null && echo "Ils sont identiques") || echo "Ils sont differents"`
   Il faut éventuellement remplacer > par 2> ci-dessus en fonction de la réponse à la question 2. On peut aussi utiliser l'option `-s` de la commande `cmp`.

   Remarque : Bien que des efforts d'uniformisation aient été faits avec la création de la norme POSIX, le comportement des commandes n'est pas une vérité absolue et dépend du système Unix et de sa configuration. C'est pour cela qu'il faut être capable de tester les commandes et de lire le manuel, qui est propre à chaque distribution.

Redirections et tubes

Exercice 4 – Enchaînement de commandes et redirections

Comme on a déjà eu l'occasion de le voir, les commandes peuvent être enchaînées grâce aux opérateurs `;`, `&&` et `||`. En combinant ces derniers et des redirections, écrivez une ligne de commande pour exécuter chacune des opérations suivantes :

- Écrire le contenu de votre répertoire de login dans le fichier `replog` et votre identifiant (`id`) dans le fichier `identifiant`.
- Si le fichier `~/toto` existe, afficher son contenu à l'écran, sinon ne rien afficher.
- Écrire dans un fichier `fic` la liste des fichiers (au sens large) de votre répertoire de login et la liste des processus lancés sur votre machine (une seule redirection).

Correction.

- `ls ~ > ~/replog ; id > ~/identifiant`
- `cat ~/toto 2> /dev/null`
- `(ls ~;ps -a) > fic`

Exercice 5 – Lancer des commandes dans un autre terminal.

Le but de cet exercice est d'exécuter des commandes depuis un terminal dans un autre terminal.

1. Lancez la commande `ps -l`. La commande elle-même est exécutée dans un processus, qui apparaît dans la liste. Identifiez son processus parent. La commande qui est à l'origine de ce processus parent est le shell dans lequel vous tapez les commandes. Cette commande s'appelle `bash` (sur d'autres systèmes, elle pourrait s'appeler `sh`, `ksh`, `tcsh`, etc.)
2. Ouvrez un autre terminal et vérifiez qu'un nouveau processus `bash` a été lancé.
3. Lancez la commande `bash` depuis un terminal et vérifiez qu'elle a généré un nouveau processus, dont le père est le précédent.
4. Le shell lit les commandes que vous tapez sur son entrée standard. Il est donc possible de rediriger cette entrée. Créez un fichier de nom `commande`, contenant uniquement la ligne suivante :
   `ps -l`
   et lancez la commande `bash` en redirigeant son entrée pour qu'elle lise le fichier `commande`.
   Sur le résultat produit, vous pourrez vérifier l'identité du processus ainsi exécuté, et l'identité de son père. Vérifiez que le processus (fils) n'existe plus.
5. À partir d'un terminal, essayez d'exécuter des commandes dans un autre terminal en utilisant un tube (rappel : les tubes ont été vus au TP5).
6. Essayez de voir le résultat des commandes dans le premier terminal. Faites en sorte de pouvoir voir aussi les messages d'erreur.

Correction. Pour les deux dernières questions, en supposant que des tubes nommés tube, f1 et f2 ont déjà été créés :

–Premier terminal : `cat > tube`
–Second terminal : `bash < tube`

Pour que le résultat apparaisse dans le premier terminal :

–Premier terminal : `cat < f1 &` puis `cat > f2`
–Second terminal : `bash < f2 > f1`

Pour récupérer aussi les messages d'erreur :

–Premier terminal : `cat < f1 &` puis `cat > f2`
–Second terminal : `bash < f2 > f1 2>&1`