Introduction aux systèmes d'exploitation (IS1)

TP 7 – enchaîner des commandes

Nous avons déjà vu comment lancer plusieurs commandes *en parallèle*, en connectant leurs sortie et entrée standard à l'aide de tubes. Il est également possible de lancer plusieurs commandes en parallèle sans redirection de leurs flots standard, simplement en les lançant en arrière-plan (sauf éventuellement la dernière) à l'aide du caractère `&`. Nous allons voir maintenant plusieurs manières de lancer des commandes *en série*, c'est-à-dire l'une après l'autre, sans attendre l'invite du shell entre les commandes.

L'enchaînement simple La première méthode consiste à utiliser le connecteur « ; » pour séparer les différentes commandes à exécuter :

$$\text{~\$>}\ commande_1\ ;\ commande_2\ ;\ \ldots\ ;\ commande_n$$

Le shell exécute alors la première commande, puis, une fois celle-ci terminée, la deuxième, et ainsi de suite jusqu'à la dernière.

Exercice 1 – *enchaînement simple* vs *exécution en parallèle*

1. Comparez l'exécution des deux instructions suivantes :
   `sleep 5 ; echo "bouh!"` et `sleep 5 & echo "bouh!"`

Pour mieux comprendre le déroulement de ces instructions, ouvrez maintenant un deuxième terminal depuis lequel vous surveillerez la naissance et la terminaison des processus créés. Pour avoir le temps de les observer, nous allons augmenter la durée des deux commandes.

2. Exécutez `sleep 200 ; xclock`. Combien de processus ont été créés ? Tuez le processus qui exécute « `sleep 200` ». Que se passe-t-il alors ? Terminez maintenant le nouveau processus.
3. Recommencez avec `sleep 200 & xclock`.
4. Exécutez maintenant `(sleep 200; xclock) &`. Déterminez la généalogie des processus créés. Tuez le processus qui exécute « `sleep 200` ». Que se passe-t-il ? Tuez le processus « `xclock` ». Que reste-t-il ?
5. Exécutez à nouveau l'instruction précédente, puis tuez le père du processus qui exécute « `sleep 200` ». Que se passe-t-il cette fois ?

Valeur de retour d'un processus 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.

Exercice 2 – *« echo $? »*

La commande « `echo $?` » affiche la valeur de retour de la précédente commande exécutée par le shell.

1. Exécutez « `ls` ». Quelle est sa valeur de retour ? Recommencez avec « `ls grosminet` ».
2. Exécutez « `sleep 2` », en laissant l'exécution aller à son terme. Quelle est sa valeur de retour ?
3. Exécutez « `sleep 100` », et interrompez le processus à l'aide de `ctrl-C`. Quelle est alors sa valeur de retour ? Recommencez en interrompant le processus à l'aide de différents signaux.

Les enchaînements conditionnés Deux autres connecteurs permettent d'exécuter des commandes les unes après les autres :

$$\tilde{}\$> \ commande_1 \ \&\& \ commande_2 \ \&\& \ \ldots \ \&\& \ commande_n$$

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

$$\tilde{}\$> \ commande_1 \ || \ commande_2 \ || \ \ldots \ || \ commande_n$$

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

Exercice 3 – *enchaînements conditionnés*

1. Exécutez `sleep 5 || xclock`, puis `sleep 5 && xclock`, sans interrompre la commande « `sleep` ». Comparez, puis tuez le(s) processus qui reste(nt).
2. Exécutez ensuite `sleep 500 || xclock`, puis `sleep 500 && xclock`, en interrompant la commande « `sleep` » par l'envoi d'un signal adapté. Comparez, puis tuez le(s) processus qui reste(nt).

Exercice 4 – *comparer deux fichiers*

Dans cet exercice, nous allons utiliser la commande « `cmp` », qui permet de tester si deux fichiers sont identiques.

1. Créez deux fichiers `pareil` et `meme` au contenu identique, et un fichier `different` au contenu différent. 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.
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 ?
3. Écrivez une séquence d'instructions qui compare deux fichiers de votre répertoire courant et affiche `les deux fichiers sont identiques` (uniquement) quand c'est le cas.
4. Même question en faisant en sorte que seule la phrase demandée s'affiche, et pas le message associé à la commande « `cmp` ».
5. Réciproquement, écrivez une commande qui affiche le message `les deux fichiers sont différents` (uniquement) quand c'est le cas.
6. Combinez ces deux séquences pour afficher la phrase correcte en fonction du résultat de la commande, sans afficher le message associé à « `cmp` ». Indice : vous pouvez délimiter une séquence d'instructions à l'aide de parenthèses.

Exercice 5 – *consulter un répertoire protégé*

1. Créez un répertoire Toto protégé en lecture contenant deux fichiers titi et tutu.
2. À l'aide d'une seule ligne de commande, affichez le contenu de Toto : pour cela, il faut modifier les droits d'accès avant l'affichage du contenu de Toto.
3. Enlevez à nouveau le droit en lecture sur Toto, et modifiez la ligne de commande pour obtenir l'affichage suivant : `le repertoire est protege, il faut modifier les droits` avant l'affichage du contenu de Toto.
4. Modifiez la ligne de commande pour que l'affichage précédent ne se produise que lorsque la modification des droits est réellement nécessaire.
5. Essayez votre ligne de commande sur un répertoire protégé d'un autre utilisateur. Dans ce cas, vous n'avez pas le droit de modifier les droits d'acces au répertoire. Modifiez votre ligne de commande pour afficher un message adéquat dans ce cas.

Exercice 6 – *le mot le plus long*

À l'aide de « `grep` », de redirections, et d'enchaînements de commandes, écrire une ligne de commande qui trouve la longueur du plus long mot présent dans un fichier (ou une borne inférieure !).

Petits scripts

Écrire des longues lignes de comandes comme à l'exercice précédent peut vite devenir fastidieux et illisible. Heureusement, il est possible d'écrire de vrais petits programmes en `bash`, couramment appelés *scripts*. En effet, `bash` est 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...

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

Tout fichier de script commence par une ligne permettant d'identifier le programme qui doit être utilisé pour l'exécuter – dans le cas d'un script `bash`, une ligne équivalente à `#! /bin/bash` (où la référence doit être remplacée par la référence du programme `bash` sur la machine concernée).

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). Pour exécuter un script, on peut taper directement dans le terminal une référence valide du fichier, ou taper la commande `bash` suivie d'un chemin du fichier.

Exercice 7 – *premiers scripts*

1. Écrire un script `enterre_tresor.sh` qui crée dans le répertoire courant un répertoire `Cachette` et dans ce répertoire un fichier `tresor` contenant le texte `tout plein de pieces d'or`. `Cachette` devra être protégé en lecture, écriture et exécution.
2. Écrire un script `fouille_cachette.sh` qui liste le contenu du répertoire `Cachette` comme cela a été vu à l'exercice 5. Le script refermera ensuite la cachette (autrement dit, protègera à nouveau le répertoire en lecture, écriture et exécution).
3. Modifier `fouille_cachette.sh` pour qu'il teste l'existence d'un fichier `tresor` dans `Cachette` et affiche soit son contenu, soit un message d'échec.

Exercice 8 – *déplacer des fichiers*

Vous trouverez sur Didel un fichier `decouvre_ile_au_tresor.sh`. Exécutez-le pour obtenir une arborescence de racine `IleAuTresor`. Chaque sous-répertoire de `IleAuTresor` contient un ou des fichiers – les trésors. Écrire un script `vole_tresors` qui déplace tous les fichiers-trésors dans votre répertoire `Cachette` (indication : il y en a 4).

Exercice 9 – *lister des fichiers par type*

Vous trouverez sur Didel un fichier `tata.tar`. Désarchivez-le pour obtenir une arborescence de racine `Tata`. Écrire un script qui liste le contenu de l'arborescence `Tata` en affichant d'abord les noms des sous-répertoires, puis ceux des fichiers ordinaires, et enfin ceux des liens symboliques (*cf.* exercices 11 et 13 du TP 5).