Programmationsystème

Corrigé du TP 2: Gestion des processus, commande `find` et introduction aux scripts

Avant de commencer, éditer le fichier *~/.proûle*. Ce fichier qui se trouve dans votre home directory est chargé automatiquement au lancement d'un shell. Placer dans ce fichier la commande suivante :

```
export PATH=/usr/xpg4/bin:/usr/X/bin:/usr/local/bin:$PATH
```

Cette commande vous permet d'avoir accès à toutes les commandes se trouvant dans les répertoires indiqués.

Commande kill

La commande `kill` permet d'envoyer à un processus un certain signal (un message très court). C'est le système qui se charge de prévenir le processus qu'il a reçu un signal. Syntaxe :

```
kill -<SIGNAL> <PID>
```

`<SIGNAL>` est une valeur numérique ou une chaîne de caractères désignant un signal. Pour connaître la liste des signaux existants, essayez la commande `kill -l`. Pour découvrir la signification des signaux lisez la page d'aide concernant les signaux : `man -s 3HEAD signal`.

Exercice :

- Lancez un éditeur `nedit` en tâche de fond (`nedit &`) ; puis exécutez la commande `ps` (par exemple avec les options `ps -u $USER`, notez le PID du processus nedit.
- Exécutez et comprenez la commande suivante :
  `ps | cut -c1-6,21-30 | grep bash`
  Le symbole guillemet inverse ` permet de demander au shell d'exécuter la commande se trouvant entre les guillemets inverses. Essayez la commande suivante :
  `echo 'ps | cut -c1-6,21-30 | grep bash'`
  Pourquoi le résultat de cette commande n'est pas identique à celui de la précédente ?

  Correction : *Un autre processus bash apparaît, il s'agit de celui utilisé pour exécuter la commande entre les guillemets inverses.*

- Créez votre premier script shell. Dans le répertoire *~/temp*, créez un fichier *monkill*, placez-y la commande : `echo 'ps | cut -c1-6,21-30 | grep $1'`.
  Donnez les droits en exécution à ce script avec la commande `chmod 744 monkill`. Exécutez le script de la manière suivante : dans le répertoire *~/temp/*, tapez `./monkill bash`.
  Dans un script, la chaîne `$0` fait référence au nom de la commande lancée, `$1` au premier argument de la commande (c.-à-d. la seconde chaîne de caractère sur la ligne de commande), ... D'autre part, `$#` donne le nombre d'arguments présents sur la ligne de commande. D'autres variables prédéfinies sont accessibles au sein d'un shell : `$*` la liste des arguments à partir de `$1`, `$$` le PID du processus courant, `$!` le PID du processus fils.
- Modifiez ce script shell pour qu'il tue les processus dont le nom correspond à la chaîne passée en paramètre.

  Correction :
  ```
  #!/bin/bash
  kill 'ps | cut -c1-6,21-30 | grep $1 | cut -c1-6'
  ```

Lancement en arrière plan (background)

Unix est un système multi-tâches. Il est donc possible de lancer une commande et que le système « nous rende la main » (c'est-à-dire que l'on peut exécuter une autre commande alors que la première n'est pas terminée). Il y a deux moyens pour lancer une tâche en *arrière plan* :

1. faire suivre la commande que l'on veut lancer par le symbole `&` ;
   Exercice : Exécutez la commande `nedit &`.
2. une fois que la commande a été lancée et que le système ne nous rend pas la main (on a lancé un processus en *avant plan*). On tape alors `Ctrl z`, ce qui a pour effet de *stopper* le processus en cours. Pour faire redémarrer ce processus en arrière plan on tape à l'invite : `bg`.
   Exercice : Exécutez la commande `nedit`. Puis `Ctrl z` et enfin `bg`. Analysez les différentes informations qui sont affichées par le shell lors de ces opérations.

On peut grouper des commandes à lancer en arrière plan en utilisant les parenthèses (ce qui a pour effet de lancer un shell qui exécute les commandes se trouvant entre parenthèses).
Exercice : Exécutez la commande `(echo cocotier;nedit;nedit) 2>/dev/null &`
Quelle est la différence avec la commande `(echo cocotier;nedit;nedit &) 2>/dev/null` ?
Quelle est la différence avec la commande `(echo cocotier;nedit & nedit) 2>/dev/null` ?

Gestion des processus en arrière plan

Quelques commandes :

`jobs` : liste des processus actifs ou stoppés (stoppés avec `Ctrl z` par exemple) ;

`fg` : exécute en avant plan (foreground) un processus stoppé ;

`bg` : exécute en arrière plan (background) un processus stoppé ;

`kill` : tue un processus s'exécutant en arrière plan (background).

Exercice : Comprenez les informations délivrées par le shell lors du lancement des commandes suivantes.

```
$ nedit &
$ jobs
$ xclock
  <Ctrl z>
$ jobs
$ kill %1
$ jobs
$ bg %2
$ fg %1
```

Commande find

Faites connaissance avec la commande `find` avec le manuel : `man find`. Essayez les commandes suivantes pour vous familiariser avec celle-ci :

```
$ find .  -name "*.c                              # noms des fichiers se terminant par 1
$ find . -type                                    # les répertoires
$ find . -name "*.c" -o -type                     # -o pour faire un OU
$ find . -name "*.c" -o -name "r*"
$ find . -print -name "*.c                        # attention a l'option -print
$ find . -name "*.c" -print -exec ls -l {} \;
$ find . -print -name "*.c" -exec ls -l {} \;
$ find . -name "*.c" -exec ls -l {} \; -print
$ find . \( -name "*.c" -o -type d \) -name "m*     # plus complique
```

Exercice : Avec la commande `find`, cherchez avec une seule commande depuis votre répertoire d'accueil :
– les fichiers se terminant par les caractères « .o » et tous les fichiers s'appelant « core ».

Correction : `find ~ -type f \( -name "*.o" -o -name core \)`

– Dans la commande `find` précédente, faites exécuter la commande `ls -la` sur tous les fichiers rencontrés.

Correction : `find ~ -type f \( -name "*.o" -o -name core \) -exec ls -la {} \;`

– Ces fichiers prennent de la place disque et souvent inutilement. On peut donc, après avoir vérifié que l'on ne s'en sert pas, lancer dans le find la commande `rm`. Mais ATTENTION ceci est à faire avec PRUDENCE.

Correction : `find ~ -type f \( -name "*.o" -o -name core \) -exec rm {} \;`

– Utilisez l'option `-mtime` pour trouver les fichiers ayant été modifiés aujourd'hui même.

Correction : `find ~ -type f -mtime 0`

Introduction au script shell (bash)

Définition de variables

– Certaines variables sont définies au sein de l'environnement associé au shell en cours d'exécution. Pour les afficher, taper `env | less`. Celles que vous utiliserez le plus sont `$HOME`, `$PATH`, `$USER`, `$SHELL`, `$PWD`, `$PS1` (format du prompt).
– L'utilisateur peut définir ses propres variables (grâce à la commande `=`). En bash, on définit par exemple la variable `moi1` avec la commande suivante dans le shell `moi1=$USER`. La commande `echo $moi1` permet d'afficher la valeur de la variable.
– Pour définir une variable on peut aussi utiliser la commande `read`, qui va lire la valeur sur l'entrée standard. Essayer : `read moi2; echo $moi2`.
– Dans de nombreuses circonstances, on souhaite que les processus fils d'un certain shell, connaisse les variables que l'on définit. En bash, on utilise pour cela la commande `export <nom de variable>`. Tapez la commande `export moi1` ; puis lancez un nouveau bash dans la fenêtre que vous avez utilisez avec la commande `bash`. Constatez que la varibale `moi1` est encore connue alors que `moi2` n'est plus connue dans ce nouveau shell.

Exercice :

– Faites un `man bash`. Recherchez la chaine de caractère PROMPTING en tapant ensuite « /PROMPT » une fois que le man est lancé. Modifier votre *.proûle* (en définissant la variable `PS1`) pour que votre prompt soit du type « `<date>-<nom de l'utilisateur>-<hôte>-<répertoire courant><saut de ligne>` ».

Correction : `export PS1='\d-\u-\h-\w\n'`