

Programmation Système

Guillaume Chanel & Florent Glück

April 09, 2025

Scripts - série 2

Introduction

Voici quelques conseils pour bien réaliser cette série d'exercices :

- n'hésitez pas à utiliser le shell pour tester des commandes, éventuellement avec des pipes ;
- décomposez votre code en fonctions ; p.ex. une commande avec plusieurs pipes est souvent plus lisible sous forme de fonction ;
- la commande `bash -x monscript` permet d'exécuter un script en traçant toutes les commandes effectuées par le script, ce qui est très utile pour le débogage.

Exercice 1 : recherche de groupes

Objectif

Le but de cet exercice est d'écrire le script `getgroups`. Celui-ci doit rechercher TOUS les groupes d'un utilisateur donné sans utiliser les commandes permettant d'obtenir ces informations directement (i.e. `groups`, `id`, etc.). Il s'agit donc de parser les contenus des fichiers `/etc/passwd` et `/etc/group`.

Dans le fichier `/etc/passwd` vous pouvez trouver le `gid` du groupe principal de l'utilisateur. Dans le fichier `/etc/group` vous pouvez trouver le nom du groupe correspondant à ce `gid`, mais également tous les autres groupes référençant le nom de l'utilisateur.

Votre script doit afficher tous les groupes sur une seule ligne. Si au moins un argument n'est pas fourni à votre script, alors il devra retourner son usage. Si votre script est utilisé avec un nom d'utilisateur non existant il ne doit rien retourner ou un indiquer un message d'erreur. L'usage et tous les messages d'erreur doivent être écrit sur l'erreur standard.

Voici quelques exemple d'utilisation du script à implémenter :

```
$ ./getgroups user
main-group other-group yet-another probably-many
$ ./getgroups
Usage:
  get-group username
$ ./getgroups nonexistentuser
nonexistinguser: user not found
```

Remarques

- Notez que les commandes `groups` et `id` restent utiles pour tester votre script.
- Attention, des utilisateurs peuvent avoir le même nom de groupe que leur login. De plus, il est possible d'avoir des utilisateurs avec des noms similaires (e.g. `chanel` et `chanel2`).

Expressions régulières

Les expressions régulières permettent de filtrer, rechercher et remplacer des séquences de caractères complexes dans du texte. Elles sont utilisées par des éditeurs de texte, des commandes (e.g. `grep`, `sed`, etc.) et des langages de programmation (e.g. module `re` de python, classe `String` en java).

La commande `grep -E expression` permet d'identifier les lignes contenant l'expression régulière indiquée.

Il existe au moins deux types d'expressions régulières : basiques (basic) et étendues (extended). Bien que les deux formes d'expressions aient beaucoup en commun elles diffèrent sur quelques points. Dans cet exercice nous ne voyons que certaines expressions régulières étendues. Pour plus d'informations vous pouvez suivre les liens suivants :

- [Ryan's tutorial](#)
- [Huge tutorial](#)

Vous pouvez également tester des expressions régulières en utilisant un des nombreux [testeur en ligne](#).

Voici quelques indications utiles pour cet exercice :

- `.` indique n'importe quel caractère ;
- `+` indique que le caractère (ou l'ensemble de caractères) précédent est présent au moins une fois ;
- `*` indique que le caractère (ou l'ensemble de caractères) précédent est présent de 0 à N fois.

Voici quelques exemples pour ces premières indications :

Expression régulière	Match	Non match
<code>Content!+</code>	<code>Content! Content!!!</code>	<code>Content</code>
<code>Content!*</code>	<code>Content! Content!!! Content</code>	
<code>123.567</code>	<code>1234567 1238567 123A567</code>	<code>12344567</code>
<code>123.*567</code>	<code>1234567 1238567 123A567 12344567 123567 12389567</code>	<code>124567</code>

Les caractères suivants permettent de définir des ensembles de caractères possibles :

- `\d`: caractère décimal
- `\s`: tout caractère d'espacement (espace, saut de ligne, tabulation)
- `\w`: alphanumérique + underscore
- `$`: caractère indiquant la fin d'une chaîne ou d'une ligne

De plus les crochets `[]` permettent de lister un ensemble de caractères possibles:

Expression régulière	Match	Non match
<code>[:,]channel(, \$)</code>	<code>:channel ,channel :channel,</code>	<code>channel :channel-hepia :,channel</code>

Grâce à ces informations vous devriez être capable de réaliser cet exercice.

À vous de jouer !

Script 2 : recherche du plus gros fichier

Le but de cet exercice est d'écrire le script `getbig`. Celui-ci doit trouver le plus gros fichier d'un répertoire de manière récursive. Le script prend donc un seul argument en entrée : le répertoire à parcourir.

Il est bien sûr interdit d'utiliser des commandes permettant de réaliser cette opération directement ou avec des pipes. En particulier, les commandes `du` et `find` sont interdites (tout au moins leur utilisation récursive).

En revanche, la commande `stat`, permettant d'obtenir la taille exacte d'un fichier doit être utilisée.

Le script doit fonctionner de manière récursive afin de trouver le fichier le plus gros, incluant une recherche dans les sous-répertoires. Uniquement la taille des fichiers réguliers doit être considérée (la taille des répertoires ne doit pas être prise en compte pour trouver le plus gros fichier).

L'usage du script ainsi que tous les messages d'erreur doivent être écrit sur l'erreur standard.

Attention : vous devez gérer le cas où des répertoires ou sous-répertoires sont vides. Les fichiers cachés peuvent être ignorés.

Voici quelques exemples de comportement du script attendus :

```
$ ./getbig
Number of argument is incorrect or parameter is not a directory
Usage:
    getbig some_dir
$ ./getbig some_dir
some_dir/subtest/biggest.txt: 3130
```