Systèmes d'exploitation

Projet: Recherche parallèle de fichiers

L2 – Université Nice Sophia Antipolis

Binôme les projets sont à faire en binôme avec une personne de votre groupe de TP

Remise le projet est à rendre en un seul fichier **rls.py** sur la boîte de dépôt de Jalon ; indiquez les noms, prénoms, et numéros d'étudiant de chacun en commentaire de votre code.

Évaluation le projet sera évalué en deux temps : d'abord sur la base de ce qui sera déposé sur Jalon à la fin de la deuxième séance de TP (semaine du 15 avril), puis sur une démo qui sera faite durant la troisième séance de TP (semaine du 22 avril). Entre la déposition et la démo il est autorisé, et même recommandé de continuer à travailler sur le projet pour corriger les derniers bugs.

Le but du projet est d'écrire un utilitaire nommé `rls` [1], qui est une version parallèle de la commande `ls -R`, autrement dit qui a pour effet d'afficher tous les fichiers du répertoire courant et de ses sous-répertoires dont le nom vérifie un motif donné. Par exemple `ls -R *.py` et `./rls \*.py` affichent la liste de tous les fichiers réguliers et répertoires dont le nom termine par `.py`. Il s'agit donc d'explorer l'arbre du système de fichiers en descendant récursivement dans tous les sous-répertoires pour lister les fichiers. Pour gagner en efficacité, on va paralléliser cette recherche en utilisant un processus par sous-répertoire. À la fin, on étend l'application avec un mode "serveur". Lorsqu'on lance `./rls -server`, l'utilitaire ouvre un service sur le port 50000. Il est possible à un autre programme (le client) de venir se connecter à ce port est de demander à rls d'effectuer une recherche de fichiers et de communiquer le résultat.

1 Expansion du nom de fichier

Commencez par récupérer le fichier `rls.py` sur Jalon. Ce fichier contient une version préliminaire du projet qui fonctionne avec un seul processus. Testez le programme en tapant dans un terminal, dans le même répertoire que celui où vous avez sauvé votre fichier `rls.py`

```
moi$ chmod u+x rls.py
moi$ touch toto.txt titi.txt
moi$ ./rls.py toto.txt
toto.txt
moi$ ./rls.py *.txt
rls.py: error: unrecognized arguments: toto.txt titi.txt
moi$
```

Comme vous pouvez le constater, `rls` ne sait pas encore gérer les *wildcards* du shell [2]

Votre première mission va être de rajouter cette fonctionnalité. Pour ce faire, vous allez simplement faire faire le travail au shell !

0. cecile.baritel-ruet@etu.univ-cotedazur ou etienne.lozes@i3s.unice.fr
1. le nom a été inventé pour ce projet
2. pour une définition des *wildcards*, relisez votre cours de système L1, ou `http://tldp.org/LDP/GNU-Linux-Tools-Summary/html/x11655.htm`

Quand vous invoquez une commande avec des wildcards depuis le shell, il commence par compléter les wildcards de toutes les manières possibles, et ensuite il exécute la commande. Par exemple, la commande `ls *.txt` est transformée par le shell en quelque chose comme `ls fichier1.txt fichier2.txt` puis le programme `ls` est exécuté. Pour cette raison, lorsque vous testerez votre programme depuis le shell, vous écrirez `./rls \*.txt` avec le backslash pour que le shell ne fasse pas le remplacement de wildcards avant de lancer votre programme.

Regardez maintenant le code du fichier `rls.py`. Repérez la variable globale `FILENAME` qui contient l'argument passé en ligne de commande à `rls`. Repérez aussi la fonction `local_ls()` qui renvoie la liste des fichiers du répertoire de travail du processus dont le nom correspond à `FILENAME`.

Vous allez modifier cette fonction pour qu'elle continue de fonctionner même si `FILENAME` contient des *wildcards*. Au lieu d'appeler `os.listdir()`, vous allez exécuter la commande

```
sh -c ls FILENAME
```

où `FILENAME` désigne la chaîne de caractères contenue dans la variable `FILENAME`. Cette commande a pour effet de lancer un nouveau shell dans lequel la ligne de commande `ls FILENAME` est évaluée. Pour exécuter cette commande, vous pourrez utiliser

```
os.execv("/bin/sh", ["sh", "-c", "ls {}".format(FILENAME)])
```

depuis un processus fils créé spécialement pour l'occasion. Mais vous voulez aussi récupérer le résultat de cette commande. Pour ce faire, vous allez rediriger la sortie standard du fils vers un tube créé pour l'occasion, et qui sera lu par le père. Vous redirigerez aussi la sortie d'erreur du fils vers le fichier `/dev/null` pour éviter d'afficher un message d'erreur chaque fois qu'aucun fichier n'est trouvé.

Indication : Vous n'avez que la fonction `local_ls()` à modifier. Vous aurez besoin des fonctions `os.fork`, `os.execv`, `os.pipe`, `os.dup2`, `os.close`[3] et `os.read`. Il vous faudra aussi convertir le *bytearray* `buf` que vous obtiendrez en lisant dans le tube avec `os.read()` en la liste `L` des chaînes de caractères des noms de fichiers renvoyés par la commande. Pour ce faire, vous pourrez utiliser l'instruction

```
L = [x for x in buf.decode().split('\n') if x != '']
```

Lorsque c'est fait, testez `./rls.py \*.txt`. Vous devriez voir s'afficher tous les fichiers `.txt` de votre répertoire et des sous-répertoires.

2 Parallélisation

On veut désormais confier chaque sous-répertoire à un processus explorateur. Par exemple, si on a l'arborescence

```
.
./rep1
./rep2
./rep2/srep1
./rep2/spre2
./rep2/srep3
```

on aura en tout 6 explorateurs. Dans la fonction `explorer`, chaque explorateur doit réaliser, dans l'ordre, les tâches suivantes :

1. aller dans le répertoire qui lui a été assigné (fonction `change_dir()`),
2. lister les fichiers du répertoire (fonction `ls_local()`),

3. n'oubliez pas de fermer les descripteurs inutiles pour éviter des blocages...

3. créer un explorateur par sous-répertoire,
   (par exemple, le processus du répertoire rep2 va créer 3 processus)
4. attendre leur terminaison et terminer lui-même proprement, avec un code sortie égal à 0 si le fichier recherché a été trouvé dans le répertoire courant ou un sous-répertoire, 1 sinon.
   Vous utiliserez la fonction `sys_exit(n)` et non `sys.exit(n)` pour pouvoir afficher des informations de debugging.

Supposons par exemple que le fichier `toto.txt` se trouve dans les répertoires `.` et `rep2/srep3`. On pourra avoir une exécution comme suit :

```
moi$ ./rls -debug toto.txt
[103210] entre dans le répertoire . et dort 1 sec
toto.txt
[103211] entre dans le répertoire rep1 et dort 2 sec
[103212] entre dans le répertoire rep2 et dort 1 sec
[103213] entre dans le répertoire srep1 et dort 1 sec
[103211] termine avec le code de sortie 1
[103214] entre dans le répertoire srep2 et dort 2 sec
[103215] entre dans le répertoire srep3 et dort 1 sec
[103213] termine avec le code de sortie 1
[103215] termine avec le code de sortie 1
rep2/srep2/toto.txt
[103214] termine avec le code de sortie 0
[103212] termine avec le code de sortie 0
[103210] termine avec le code de sortie 0
```

Indication : Vous n'aurez que la fonction `explorer` à modifier, et vous aurez besoin de `os.fork` et `os.wait`. Par rapport à la partie 1, cela devrait être plus facile, mais récupérez bien toutes les terminaisons de processus.

3 Recherche rapide

Vous allez maintenant ajouter un mode *recherche rapide* à votre utilitaire, que l'on activera avec l'option `-first_match`. En mode recherche rapide, on veut rapidement savoir si le fichier recherché se trouve dans au moins un sous-répertoire, sans nécessairement lister tous les sous-répertoires qui le contiennent. Ainsi, dès qu'un processus a trouvé un répertoire qui contient le fichier recherché, il va informer tous les autre processus de sorte que l'utilitaire puisse s'arrêter promptement et proprement. Plus précisément, chaque explorateur va réaliser les tâches suivantes :

1. terminer immédiatement, avec le code de sortie 0, sans créer de fils pour les sous-répertoires si le fichier a été trouvé dans le répertoire local
2. vérifier si un de ses fils a trouvé le fichier ; si c'est le cas, envoyer un signal SIGUSR1 à tous les fils restants pour leur dire de cesser les recherches, attendre leur terminaison, puis terminer proprement avec le signal 0.
3. en cas de réception d'un signal SIGUSR1, le propager à tous les fils, attendre leur terminaison, puis terminer proprement avec le code de sortie 2.

Exemple : supposons que le fichier `toto.txt` se trouve dans le répertoire rep1/ (et nulle part ailleurs). On pourra avoir l'exécution suivante :

```
moi$ ./rls -first_match -debug toto.txt
[103210] entre dans le répertoire . et dort 1 sec
```

```
[103211] entre dans le répertoire rep1 et dort 2 sec
[103212] entre dans le répertoire rep2 et dort 1 sec
[103213] entre dans le répertoire srep1 et dort 1 sec
[103214] entre dans le répertoire srep2 et dort 1 sec
[103215] entre dans le répertoire srep3 et dort 1 sec
rep1/toto.txt
[103211] termine avec le code de sortie 0
[103213] termine avec le code de sortie 2
[103214] termine avec le code de sortie 2
[103215] termine avec le code de sortie 2
[103212] termine avec le code de sortie 2
[103210] termine avec le code de sortie 0
```

Indication : Vous allez devoir changer le code de la fonction `main` pour installer un handler de signaux dans le processus père (il sera hérité pour les fils) lorsque l'option `first_match` est activée. Il faudra aussi revoir le code de la fonction `explorer`, et vous aurez peut-être besoin de mettre dans une variable globale la liste des pids des fils encore actifs (à vous de comprendre pourquoi...)

BONUS (optionnel) : pour terminer encore plus rapidement, il serait plus efficace qu'un fils qui est au courant que le fichier a été trouvé prévienne immédiatement son père en lui envoyant le signal SIGUSR2, et que celui-ci prenne le temps de faire terminer ses propres fils proprement.

4 Le mode serveur

Votre dernière mission va être de rajouter la gestion de l'option `-server`. En mode serveur, rls se met en attente de requêtes sur le port 5000. Pour chaque connection acceptée, il crée un processus de service. Le processus de service attend sur la socket de service que le client lui envoie une valeur pour FILENAME. Une fois FILENAME connu, la recherche se fait comme précédemment, à ceci près que l'utilitaire n'affiche pas les fichiers trouvés, il les envoie sur la socket de service, et c'est au client d'en faire bon usage.

On pourra tester le programme en utilisant deux terminaux et en reproduisant l'exécution suivante [4]

```
moi$ ./rls -server
```

```
moi$ nc localhost 50000
toto.txt
toto.txt
rep2/srep2/toto.txt
moi$
```

BONUS (optionnel) : pour le moment le serveur ne s'arrête jamais, il faut le tuer avec un signal pour l'arrêter, et on n'est pas sûr que les processus de service ont bien pris fin. Vous pourrez si vous le souhaitez écouter en même temps la socket d'écoute et l'entrée standard et déclencher une terminaison propre du serveur lorsqu'une entrée clavier est effectuée.

4. la partie de couleur bleue correspond à ce qui est affiché, et celle en noir ce qui est tapé au clavier.