Projet `leaSH` (Learning SHell)

RS : Réseaux et Systèmes

L'objectif de ce projet est d'écrire un shell Unix un peu particulier, puisqu'il permet de poser des défis pédagogiques à l'utilisateur. Ce shell pédagogique sera utilisé lors des cours d'introduction en début de 1A pour enseigner les bases d'utilisation d'Unix et les commandes principales (cd, cp, cat, mkdir, grep, sed, etc).

Pour réaliser cet environnement, on sera amené à utiliser la plupart des appels systèmes vus en cours comme `fork(2)` et `execvp(3)` (pour lancer des commandes), `pipe(2)` et `dup2(2)` (pour modifier stdin et stdout des commandes lancées), `read(2)` et `write(2)` (pour communiquer avec les commandes lancées), `sigaction` (pour manipuler les signaux). L'appel système `chdir(2)` sera également utile pour implémenter la commande `cd`. Il est également possible que l'appel système `fcntl(2)` s'avère utile.

1 Informations générales

1.1 Comment travailler

Apprendre à travailler en groupe fait partie des objectifs pédagogiques de ce projet. Il vous est donc demandé de travailler en binôme ou par groupes de trois si vous le souhaitez (mais nous tiendrons compte de la taille des groupes lors de la notation – implémentez plus d'extensions pour compenser). Il est **interdit** de travailler seul. Vous êtes libre de suivre ou non la stratégie d'implémentation proposée, mais il est **indispensable** que votre projet fonctionne sous linux et soit implémenté en C.

1.2 Évaluation de ce projet

Soutenances. Des soutenances pourront être organisées. Vous devrez nous faire une démonstration de votre projet et être prêts à répondre à toutes les questions techniques sur l'implémentation de l'application.

Rapport. Vous devez rendre un mini-rapport de projet (5 pages maximum plus page de garde, format pdf). Vous y détaillerez vos choix de conception, les difficultés auxquelles vous avez été confronté, et comment vous les avez résolues. Vous indiquerez également le nombre d'heures passées sur les différentes étapes de ce projet (conception, codage, tests, rédaction du rapport, etc.) par membre du groupe.

Section «Remerciements». Votre rapport doit contenir quelque chose comme la mention suivante : «Nous avons réalisé ce projet sans aucune forme d'aide extérieure» ou bien «Nous avons réalisé ce projet en nous aidant des sites webs suivants (avec une liste de sites, et les informations que vous avez obtenu de chaque endroit)» ou encore «Nous avons réalisé ce projet avec l'aide de (nommer les personnes qui vous ont aidé, et indiquez ce qu'ils ont fait pour vous)». Si la liste est trop longue, vous pouvez la placer en annexes (au delà des cinq pages de votre rapport).

Aider et se faire aider est encouragé, mais l'honnêteté académique vous impose de lister vos aides. Tout manquement à cette règle est une tricherie et sera sanctionné comme il se doit. Par se faire aider, on entend : avoir une discussion sur le design du code, y compris sur des détails techniques et/ou les structures de données à mettre en œuvre. Par tricher, on entend : copier ou recopier du code ou encore lire le code de quelqu'un d'autre pour s'en inspirer. Nous testerons l'originalité de votre travail par rapport aux autres projets rendus [1], et il est difficile de défendre en soutenance un projet que l'on n'a pas écrit.

Section «Licence». Vous préciserez la licence logicielle sous laquelle vous souhaitez diffuser votre code et vos niveaux. Cet élément n'entrera nullement dans la notation, et vous êtes libre de refuser que votre travail soit diffusé ou bien de souhaiter qu'il soit réutilisé par exemple en 1A ou plus largement.

«Rendu» du projet. Vous devez utiliser un dépôt SVN sur la forge de l'école (`http://forge.telecomnancy.univ-lorraine.fr/`). Créez votre projet comme un sous-projet de *RS 2014* (dans *Projets divers*). Votre projet doit être privé pour ne pas que les autres binômes puissent y accéder. L'identifiant de votre projet doit être de la forme `rs2014-login1-login2` (où `login1` et `login2` sont les deux logins des membres du binôme). Ajoutez *Martin Quinson* (login : `quinson5`) aux développeurs de votre projet.

Le projet sera récupéré directement sur votre dépôt SVN. **Dès votre binôme constitué** et votre projet créé, envoyez par mail à `martin.quinson@loria.fr` vos noms, prénoms, logins, l'identifiant de votre projet ainsi que le chemin d'accès **SVN** à votre projet. Ce dépot SVN devra contenir :

- `AUTHORS` : liste des noms, prénoms et login des membres du groupe (une personne par ligne) ;
- Un `Makefile` compilant votre projet en créant un fichier exécutable nommé `leaSH` ;
- `rapport.pdf` : votre rapport au format PDF.

1. `http://theory.stanford.edu/~aiken/moss/`

Batterie de tests. Une partie de la note sera attribuée de manière semi-automatique par une batterie de tests. Vous perdrez des points si votre programme ne se comporte pas comme attendu. En particulier, votre programme doit renvoyer un code d'erreur de 0 en cas de succès et seulement dans ce cas.

Niveaux de jeu. Quelques niveaux de jeu seront diffusés depuis la page web du cours, mais il est attendu que vous rendiez vos propres niveaux avec votre projet. Des niveaux nombreux, originaux, intéressant pédagogiquement et amusant à jouer vous rapporteront beaucoup de points de bonus.

2 Description du projet leaSH

Vous devez écrire un programme leaSH, qui est une sorte de shell un peu particulier. Il met ses utilisateurs dans une situation prédéfinie par un fichier de configuration, et vérifie qu'ils parviennent à réaliser les tâches attendues d'eux. Plus précisément, c'est un shell permettant d'écrire des exercices d'apprentissage de la ligne de commande Unix. Il se rapproche donc de projets tels que Terminus, Bashy ou bandit wargame, mais il est implémenté comme un shell classique tel que bash ou zsh (en plus simple).

Si vous ne comprenez pas de quoi il s'agit, rendez vous à la section 2.4 qui présente un exemple d'usage, puis revenez ici ensuite.

Votre programme leaSH doit accepter un seul argument sur la ligne de commande. Il s'agit du nom d'une archive tar.gz décrivant le niveau de jeu à lancer. Si l'argument désigne un fichier ne pouvant être lu, votre programme doit s'arrêter avec un message et un code d'erreur appropriés.

2.1 Contenu des archives décrivant les niveaux

Chaque fichier que le leaSH accepte en argument décrit un niveau de jeu. Il doit s'agir d'une archive tar, éventuellement compressée, contenant l'arborescence de fichiers constituant le niveau. En plus des fichiers normaux avec lesquels les joueurs vont interagir, l'archive contient un un fichier nommé meta à la racine de l'arborescence, qui donne des informations sur le niveau (voir la section suivante).

Au démarrage d'un niveau, leaSH crée un répertoire temporaire, et y décompresse les fichiers du niveau. Il lit les informations supplémentaires depuis le fichier meta, puis exécute les commandes tapées par l'utilisateur jusqu'à ce que celui-ci parvienne à réaliser les tâches demandées par le niveau.

2.2 Syntaxe du fichier meta

Les lignes d'un fichier de description peut être de différents types, en fonction de leur premier caractère. Le second caractère doit être un espace (ignoré par leaSH). Ces lignes peuvent être séparées par des lignes blanches, qui seront également ignorées.

Caractère	*Signification*
'#'	Commentaire à ignorer.
'$'	Commande shell autorisée.
'>'	Chaîne que l'utilisateur doit faire afficher pour passer le niveau.

Pour des raisons de sécurité, leaSH n'accepte de démarrer que les commandes ayant été explicitement autorisées par des lignes du fichier meta commençant par $.

2.3 Actions réalisées par leaSH

Une fois démarré, leaSH doit ressembler le plus possible à un shell classique tel que bash. Il n'est certes pas raisonnable d'espérer réimplémenter tout bash en 30h de projet, mais vous devez quand même tendre vers cet objectif. En particulier, le comportement en cas d'erreur (commande inexistante, droits manquant pour exécuter une commande, signaux reçus, etc) devrait ressembler autant que possible à celui de bash, même si une copie à l'identique n'est pas toujours possible.

La plus grande différence entre leaSH et un shell normal est que leaSH "écoute" les affichages des commandes qu'il lance afin de détecter quand la phrase secrète du niveau est affichée. Quand c'est le cas, leaSH affiche un message adapté et se termine.

Commande spéciale : cd. Cette commande ne peut pas être traitée de la même façon que les autres, puisqu'il n'existe pas de programme `cd` à démarrer. À la place, il faut changer le répertoire de travail de `leaSH`, c'est-à-dire celui dans lequel les processus fils démarrent les commandes. `leaSH` peut changer son propre répertoire de travail (avec l'appel `chdir(2)`), ou bien stocker ce répertoire dans une variable adéquate. Dans tous les cas, il doit tenter d'empêcher les joueurs de sortir du répertoire contenant le niveau de jeu. En particulier, un `cd ..` lancé à la racine du niveau de jeu devrait être détecté et lever un message d'erreur.

2.4 Exemple de niveau

L'archive `simple.tgz`, disponible sur le site du cours, contient un niveau très simple. Cette archive ne contient que les quatre fichiers suivants : `meta`, `README`, `file1.txt` et `file2.txt`.

meta
```
1 # Exemple simple de niveau
2 $ ls
3 $ cat
4 > Bravo! C'est ici!
```

file1.txt
```
Ce n'est pas ici.
```

file2.txt
```
Bravo! c'est ici!
```

README
```
Cherchez la phrase secrète
dans les fichiers.
```

Exemple de session
```
bash $ ./leash simple.tgz
Welcome to leaSH. You are playing simple.tgz

Use 'ls' to look around, 'cat' to explore the file content,
'cd' to move around and 'man' to learn about the commands
$ ls
file1.txt
file2.txt
README
$ cat README
Cherchez la phrase secrète
dans les fichiers.
$ cat file1.txt
Ce n'est pas ici
$ cat file2.txt
Bravo! C'est ici!

Congratulation! You solved that level!
The magic key was indeed "Bravo! C'est ici!".
bash $
```

Dans le fichier `meta`, la première ligne est ignorée puisqu'elle commence par un #. Les lignes 2 et 3 précisent que seules les commandes `ls` et `cat` sont utilisables (l'utilisateur aura un message d'erreur s'il tente de lancer une autre commande). La dernière ligne du fichier `meta` indique que l'utilisateur doit faire en sorte que l'une des commandes lancées affiche la chaîne de caractères «`Bravo! c'est ici!`» pour que `leaSH` considère le niveau comme gagné. Les autres fichiers seront placés à la disposition de l'utilisateur.

Une session de travail typique est donnée dans la colonne de droite. Il est attendu dans ce niveau que l'utilisateur explore son environnement avec `ls`, lise la consigne placée dans `README` avec `cat`, puis affiche chaque fichier tour à tour jusqu'à trouver la chaîne secrète.

3 Stratégie d'implémentation possible

Pour information, voici comment j'ai procédé lorsque j'ai implémenté le projet. Vous êtes libre de suivre ce cheminement ou non, mais vous devez implémenter toutes les fonctionnalités décrites ici. Notez qu'il n'est pas suffisant d'implémenter ces items, de faire un bon rapport et d'avoir quelques niveaux supplémentaires pour espérer avoir une très bonne note au projet. Il faut également implémenter certaines des extensions présentées à la section suivante pour cela.
N'oubliez pas de commenter votre code au fur et à mesure. Dans la mesure du possible, écrivez votre code en anglais (commentaires et identifiants), car les équipes de développement sont de plus en plus souvent multiculturelles. Vous pouvez utiliser la fonction `system(3)` pour réaliser certaines des étapes du programme sans les implémenter, mais votre note risque de s'en ressentir si vous abusez de cette ruse.

1. Mettez en place votre projet en écrivant un fichier C principal et le makefile qui le compilera sous le nom `leaSH`.
2. Écrivez le code mettant en place un niveau de jeu dont le nom est passé en paramètre. Il faut créer un sous-répertoire vide (mkdir(2)), s'y déplacer, et lancer le programme `tar` pour extraire le contenu de l'archive. Testez votre code.
3. Il faut ensuite lire le fichier `meta`.
   — Ouvrez le fichier en lecture avec `fopen(3)`, puis lisez son contenu ligne à ligne avec la fonction `getline(3)`. Vous sauvegarderez la liste des commandes autorisées dans un tableau de chaînes de caractères (assurez vous que votre tableau est assez grand avant d'ajouter des choses), et la chaîne secrète dans une variable `char*`.
   — Attention, il faut recopier les valeurs que l'on lit avec getline. Utilisez par exemple strdup(3).
   — Après l'avoir lu, il convient d'effacer le fichier meta (avec `unlink(2)`) afin que l'utilisateur ne puisse pas le lire pour tricher.

4. Il est temps d'écrire la boucle principale du programme : le programme attend une commande en entrée standard, l'exécute dans un processus fils dans le répertoire du niveau, et récupère la sortie.
   — On peut réutiliser `getline(3)` pour lire les commandes de l'utilisateur en l'utilisant sur le descripteur `stdin` (il est de type `FILE*`). Ce descripteur est prédéfini dans le système, vous n'avez rien à faire avant de l'utiliser. Pour quitter le programme, il suffit de taper Ctrl-D à l'invite. Cela fonctionne également avec les autres shells, puisque le caractère Ctrl-D signifie "fin de fichier".
   — Découpez la chaîne obtenue en éléments séparés par des espaces (avec `strtok(3)`). Par exemple, si vous avez lu la chaîne "cat README", il faut construire le vecteur de chaînes suivant :
   `arguments = { "cat", "README", NULL}`
   On supposera que la chaîne lue ne contient ni guillemet, ni tube, ni redirections de flots.
   — Faites en sorte de lancer la commande obtenue dans un processus fils. Pensez à placer votre processus fils dans le bon répertoire (avec `chdir(2)`) avant d'exécuter la commande.
   — Ajoutez un tube pour que le père puisse lire la sortie standard du fils. Une fois son fils lancé, le père lit donc le contenu du tube, affiche ce qu'il lit (pour que le joueur humain voit le texte sur son terminal) et place ce texte dans une chaîne de caractères dédiée.
   — Quand il n'y a plus rien à lire sur le tube, c'est que le fils a terminé (le père devrait utiliser `waitpid(2)`). Ensuite, le père compare ce qu'il a lu à la chaîne secrète du niveau. Si cela correspond, il se termine après avoir affiché un message de félicitations au joueur si c'est gagné.

   Le gros œuvre est terminé ; vous devriez maintenant pouvoir passer le niveau `simple.tgz` fourni.
5. Ajoutez le code testant si la commande est autorisée ou non.
6. Implémentez la commande `exit`, qui termine le shell.
7. Implémentez la commande `cd`. Le plus simple est de ne pas faire le chdir(2) immédiatement, mais de garder une variable `cwd` à jour (cwd=current working directory). C'est le processus fils qui fait le chdir avant l'exec.

   Il n'est pas simple de réussir à mettre cette variable à jour correctement dans tous les cas, en vérifiant qu'on ne remonte jamais plus haut que le répertoire racine de l'exercice. La fonction `strtok(3)` s'avère précieuse pour découper l'argument de la commande `cd` et évaluer les changements de répertoires les uns après les autres, tandis que `strrchr(3)` permet de retrouver la dernière occurrence d'un caractère dans une chaîne (par exemple de '/' dans la variable cwd).

   Vérifiez que les appels suivants fonctionnent : cd .. ; cd subdir/../.. ; cd subdir/.. ; cd subdir///..

   `cd` sans argument devrait revenir à la racine de l'exercice.
8. Implémentez la commande `pwd`, qui affiche le `cwd`.
9. Vérifiez avec l'appel `stat` que vous pouvez accéder au répertoire voulu, c'est-à-dire que le répertoire existe, et vous avez le droit d'exécution dessus. Il faut vérifier cela pour chaque élément du chemin traversé.
10. Faites le nécessaire pour que l'utilisateur puisse interrompre le processus fils avec un Contrôle-C.
11. Implémentez la gestion des redirections (< et >) dans la commande. Le processus fils doit ouvrir le fichier en question et utiliser dup2(2). Un processus dont la sortie est redirigée dans un fichier n'est pas écouté par `leaSH`. On ne gagne pas s'il affiche la chaîne secrète, puisque cet affichage ne se fait pas sur la sortie habituelle.
12. Implémentez la gestion des tubes entre différentes sous-commandes. Seule la dernière commande du tube est écoutée par `leaSH` et peut donc afficher la chaîne secrète avec succès.
13. Vérifiez que votre programme passe avec succès les tests fournis sur la page web du cours, s'ils sont déjà disponibles. Faites également vos propres tests pour vous assurer du bon comportement de votre programme.
14. Faites en sorte que votre programme soit propre à l'exécution : descripteurs fermés, pas de processus zombies, pas de fuite mémoire trop grave, etc. Ces nettoyages devraient être faits même si l'exécution de `leaSH` s'arrête brutalement à cause d'un problème.
15. Nettoyez votre code avant de le rendre : supprimez le code devenu inutile (si besoin), et vérifiez que les commentaires correspondent bien au code écrit.
16. Écrivez des niveaux pour notre jeu. Vous pouvez vous inspirer du cours (et des TP) de shell que vous avez eu dans le passé, ou inventer complètement en vous inspirant des exemples fournis. L'intérêt pédagogique et ludique sont importants tous les deux, même s'il est souvent difficile de combiner les deux en shell :)

4 Aller plus loin

Cette section contient quelques idées pour les curieux souhaitant aller plus loin. Implémenter quelques-unes d'entre elles est indispensable pour prétendre avoir 20 au projet.

Si vous avez d'autres idées d'extensions, vous êtes les bienvenus pour les implémenter (et les documenter dans votre rapport). Il peut être raisonnable de me demander mon avis avant de vous lancer dans des extensions trop ambitieuses. Dans l'absolu, il serait possible de faire de `leaSH` un vrai shell complet, avec la gestion des variables, des sous-shell, ainsi que la possibilité d'écrire des scripts contenant `if`, `while`, `case`, etc. Ce serait très pratique dans le cadre de l'enseignement aux 1A, mais cela dépasse assez largement le cadre d'un projet 2A.

1. Rajoutez la possibilité de naviguer dans l'historique des commandes avec les flèches directionelles comme dans un vrai shell (avec la fonction `readline(3)`).
2. Implémentez la possibilité de chaîner plusieurs commandes avec les symboles `&&` et ||. Il s'agit de tests booléens classiques : `cmd1 && cmd2` n'exécutera `cmd2` que si `cmd1` a réussi, c'est à dire si son code de retour est égal à 0.
3. Encodez (et décodez) le fichier meta pour que la solution ne soit pas visible si on l'ouvre manuellement avant de lancer `leaSH`.
4. Implémentez la gestion des guillemets pour protéger les espaces de la ligne de commande, ainsi que du caractère '\'.
5. Augmentez la robustesse de votre programme, en gérant le cas où le chemin passé à `cd` est un lien (par exemple un lien pointant en dehors du répertoire du niveau), en vérifiant que le joueur a le droit d'exécution sur les commandes à venir (même si le joueur n'est pas le propriétaire du fichier), commandes arrêtées par des signaux, et tous les autres cas d'erreur potentiel auxquels vous pensez. Documentez ces cas dans votre rapport.
6. Implémentez le développement de la ligne de commande (*globbing* en anglais), qui permet de développer les motifs génériques tels que `*` ou `*.o` dans les noms de fichiers. La fonction `glob(3)` est faite pour cela.
7. Implémentez la gestion des backquotes. Le texte placé entre backquotes doit être exécuté comme une commande indépendante, et le texte que cette commande affiche doit être placé à la place des backquotes. Cela permet des usages comme `rm ‘ls‘`, qui est équivalent à `rm *`.
8. Réalisez un vrai jeu qui comporte plusieurs niveaux, au lieu de devoir lancer l'application avec le nom du niveau à essayer. Il s'agit à minima d'offrir une interface (en ascii art voire avec `ncurses(3)`) permettant de choisir le niveau auquel on veut jouer parmi ceux déjà débloqués. On peut également augmenter le format du fichier meta afin de permettre plusieurs chaînes secrètes par niveau, chacune ≪téléportant le joueur≫ vers un niveau différent. Cela peut permettre de réaliser une aventure cohérente comme celle de Terminus.
9. Implémentez la complétion des commandes, qui permet de compléter les mots avec la touche `TAB`.
10. Ajoutez le nécessaire pour la gestion du terminal et des travaux, avec les commandes `fg` et `bg`, et le signal `SIGTSTP` (Ctrl+Z). La page suivante est précieuse pour cela : `http://www.gnu.org/software/libc/manual/html_node/Implementing-a-Shell.html`
11. Remplacez la restriction sur les commandes autorisées par une sécurisation efficace basée sur `chroot(2)`. Cette méthode est bien plus sûre, mais elle impose que `leaSH` s'exécute avec les droits du super-utilisateur (ce qui n'est pas raisonnable pour un projet étudiant).

    Attention, il ne suffit pas de lancer l'appel chroot pour sécuriser la situation. Référez-vous par exemple à la page suivante : `https://filippo.io/escaping-a-chroot-jail-slash-1/`

    Un montage `aufs` peut éviter de devoir copier dans le chroot l'intégralité des binaires et bibliothèques utilisées. Voir par exemple ;
    `http://calimeroteknik.free.fr/blag/?article11/fonctionnement-live`
12. Une façon très différente d'obtenir des points de bonification pour le projet est de contribuer à l'encyclopédie Wikipédia sur des sujets en rapport avec le cours. Indiquez dans le rapport vos contributions que vous aurez pris soin de faire sous votre identité réelle.