Systèmes d'exploitation

Projet - Shell

Rendu

La date limite de rendu du projet est le **vendredi 2 juin à 23h59**.

Le projet doit être réalisé **individuellement**.

Développez votre programme en utilisant GitHub classroom. Votre travail devra être soumis à l'adresse suivante: https://classroom.github.com/a/HdkhcjpH.

Vue d'ensemble

Dans ce projet, vous allez construire un simple shell en utilisant le **langage C**. Cet interpréteur de lignes de commande (CLI) devra fonctionner de la manière suivante de base : lorsque vous tapez une commande (en réponse à l'invite), l'interpréteur crée un processus enfant qui exécute la commande que vous avez saisie, puis demande à l'utilisateur d'entrer d'autres données lorsqu'il a terminé.

L'interpréteur de commandes que vous allez créer sera similaire, mais plus simple, que ceux que vous exécutez tous les jours sous Unix.

Spécifications du programme

Votre interpréteur de commandes de base, appelé `dash` (abréviation de Dauphine shell), est essentiellement une boucle interactive : il affiche de façon répétée l'invite `dash>` (notez l'espace après le signe supérieur à), analyse l'entrée, exécute la commande spécifiée sur cette ligne d'entrée, et attend que la commande se termine. Cette opération est répétée jusqu'à ce que l'utilisateur tape `exit`. Le nom de votre exécutable final devra être `dash`.

L'interpréteur de commandes peut uniquement être invoqué sans argument. Tout autre argument résulte en une erreur.

Voici la méthode sans argument :

```
$ ./dash
dash>
```

À ce stade, `dash` est en cours d'exécution, et prêt à recevoir des commandes.

Vous devez structurer votre interpréteur de commandes de manière à ce qu'il crée un processus pour chaque nouvelle commande (à l'exception des *commandes intégrées*, discutées ci-dessous).

Votre interpréteur de commandes devra être capable d'analyser une commande et d'exécuter le programme correspondant à la commande.

Par exemple, si l'utilisateur tape `ls -la /tmp`, votre interpréteur de commandes devra exécuter le programme `/bin/ls` avec les arguments `-la` et `/tmp` (comment l'interpréteur de commandes sait-il qu'il doit exécuter `/bin/ls` ? C'est ce qu'on appelle le **path** de l'interpréteur de commandes ; plus d'informations à ce sujet ci-dessous).

Structure

Shell de base

L'interpréteur de commandes est conceptuellement simple : il s'exécute dans une boucle et demande à plusieurs reprises des données qui lui indiquent la commande à exécuter. Il exécute ensuite cette commande. La boucle continue indéfiniment, jusqu'à ce que l'utilisateur tape la commande intégrée `exit`, suite à laquelle il se termine. C'est tout !

Pour lire les lignes d'entrée, vous devrez utiliser `getline()`. Cela vous permettra d'obtenir facilement des lignes d'entrée arbitrairement longues.

Pour analyser la ligne d'entrée en éléments constitutifs, vous pouvez utiliser `strsep()`. Lisez attentivement la page du manuel pour plus de détails.

Pour exécuter des commandes, utilisez `fork()`, `exec()`, et `wait()/waitpid()`.

Vous noterez qu'il existe une variété de commandes dans la famille `exec` ; pour ce projet, vous devez utiliser `execv`. Vous ne devez **pas** utiliser l'appel de fonction de la bibliothèque `system()` pour lancer une commande. Rappelez-vous que si `execv()` réussit, la fonction ne renverra pas ; si elle renvoie, c'est qu'il y a eu une erreur (par exemple, la commande n'existe pas).

Path

Dans notre exemple ci-dessus, l'utilisateur a tapé `ls` mais l'interpréteur de commandes sait qu'il doit exécuter le programme `/bin/ls`. Comment votre interpréteur de commandes sait-il cela ?

Il s'avère que l'utilisateur doit spécifier une variable **path** pour décrire l'ensemble des répertoires à rechercher pour les exécutables. L'ensemble des répertoires qui composent le chemin sont parfois appelés le *search path* de l'interpréteur de commandes. La variable contient la liste de tous les répertoires à rechercher, dans l'ordre, lorsque l'utilisateur tape une commande.

Important : notez que l'interpréteur de commandes lui-même n'implémente pas `ls` ou autres commandes (à l'exception des commandes intégrées). Tout ce qu'il fait est de trouver ces exécutables dans l'un des des répertoires spécifiés par `path` et de créer un nouveau processus pour les exécuter.

Pour vérifier si un fichier particulier existe dans un répertoire et s'il est exécutable, utilisez l'appel système `access()`. Par exemple, lorsque l'utilisateur tape `ls`, et que le chemin est défini pour inclure à la fois `/bin` et `/usr/bin`, essayez `access("/bin/ls", X_OK)`. Si cela échoue, essayez "/usr/bin/ls". Si cela échoue aussi, c'est une erreur.

Le chemin initial de votre interpréteur de commandes doit contenir un répertoire : `/bin`

Note : La plupart des interpréteurs de commandes vous permettent de spécifier un binaire sans utiliser de chemin de recherche, en utilisant des **chemins absolus** ou des **chemins relatifs**. Par exemple, un utilisateur pourrait taper le **chemin absolu** `/bin/ls` et exécuter le binaire `ls` sans qu'un chemin de recherche ne soit nécessaire. Un utilisateur peut également spécifier un **chemin relatif** qui commence par le répertoire de travail actuel et qui directement l'exécutable, par exemple `./main`. Dans ce projet, vous **n'avez pas pas** à vous préoccuper de ces fonctionnalités.

Commandes intégrées

Chaque fois que votre interpréteur de commandes accepte une commande, il doit vérifier si cette commande est une **commande intégrée** ou non. Si c'est le cas, elle ne doit pas être exécutée comme les autres programmes. Au lieu de cela, votre interpréteur de commandes invoquera votre implémentation de la commande intégrée intégrée. Par exemple, pour implémenter la commande intégrée `exit`, vous devez simplement appeler `exit(0);` dans votre code source, ce qui aura pour effet de sortir de l'interpréteur de commandes.

Dans ce projet, vous devriez implémenter `exit`, `cd`, et `path` en tant que commandes intégrées.

- `exit` : lorsque l'utilisateur tape `exit`, l'interpréteur de commandes devra simplement appeler l'appel système `exit` avec 0 comme paramètre. C'est une erreur de passer des arguments à `exit`.
- `cd` : `cd` prend toujours un argument (0 ou >1 arguments devraient être signalés comme une erreur). Pour changer de répertoire, utilisez l'appel système `chdir()` avec l'argument fourni par l'utilisateur ; si `chdir` échoue, c'est aussi une erreur.
- `path` : la commande `path` prend 0 ou plusieurs arguments, chaque argument étant séparé des autres par des espaces. Une utilisation typique serait la suivante : `dash> path /bin /usr/bin`, ce qui ajouterait `/bin` et `/usr/bin` au chemin de recherche du shell. Si l'utilisateur définit un chemin vide, l'interpréteur de commandes ne pourra pas exécuter de programmes. La commande remplace toujours l'ancien chemin par le nouveau chemin spécifié.

Redirection

Parfois, un utilisateur du shell souhaite utiliser le contenu d'un fichier comme entrée d'un autre programme. En général, l'interpréteur de commandes fournit cette fonctionnalité avec le caractère `<`. Formellement, cela s'appelle la redirection de l'entrée standard. Votre shell devra également inclure cette fonctionnalité.

Par exemple, si un utilisateur entre la commande `wc -l < file.txt`, il obtiendra le décompte du nombre de lignes dans le fichier. L'entrée standard du programme `wc` ayant été remplacée par le fichier `file.txt`.

Le format exact de la redirection est une commande (et éventuellement quelques arguments) suivie du symbole de redirection `>` et d'un nom de fichier. Plusieurs opérateurs de redirection ou plusieurs fichiers à droite du symbole de redirection sont des erreurs.

L'appel système `dup2()` vous sera utile pour implémenter cette fonctionnalité.

Note : ne vous préoccupez pas de la redirection pour les commandes intégrées (par exemple, nous ne testerons pas ce qui se passe lorsque vous tapez `path > file.txt`).

Erreurs du programme

Vous devez afficher ce **seul et unique message d'erreur** chaque fois que vous rencontrez une erreur, quel qu'en soit le type.

```
char error_message[30] = "An error has occurred\n";
write(STDERR_FILENO, error_message, strlen(error_message));
```

Le message d'erreur doit être écrit sur stderr (erreur standard), comme indiqué ci-dessus.

Après la plupart des erreurs, votre shell *continue simplement le traitement* après avoir affiché le seul et unique message d'erreur.

Il y a une différence entre les erreurs que l'interpréteur de commandes détecte et celles que le programme détecte. Votre interpréteur de commandes doit détecter toutes les erreurs de syntaxe spécifiées dans ce document. Si la syntaxe de la commande semble parfaite, vous pouvez simplement exécuter le programme spécifié. S'il y a des erreurs liées au programme (par exemple, arguments invalides pour `ls` lorsque vous l'exécutez), l'interpréteur de commandes n'a pas à s'en préoccuper (au contraire, le programme affichera ses propres erreurs et quittera).

Conseils

Souvenez-vous de faire fonctionner les **fonctionnalités de base** de votre interpréteur de commandes avant de vous préoccuper de toutes les conditions d'erreur et des cas limites. Par exemple, commencez par une seule commande (probablement d'abord une commande sans argument, comme `ls`).

Ensuite, ajoutez des commandes intégrées. Puis, essayez de travailler sur la redirection.

A un moment donné, vous devriez vous assurer que votre code est robuste aux espaces de différentes sortes, y compris les espaces () et les tabulations (`\t`). En général, l'utilisateur devrait pouvoir mettre des quantités variables d'espaces avant et après les commandes, les arguments et divers opérateurs. Cependant, notez queles opérateurs (redirection et commandes parallèles) ne nécessitent pas d'espace blanc.

Vérifiez les codes de retour de tous les appels système dès le début de votre travail. Cela permettra souvent de détecter des erreurs dans la manière dont vous invoquez ces nouveaux appels système.

Testez votre programme. Soumettez lui de nombreuses entrées différentes et assurez-vous que l'interpréteur de commandes se comporte bien.

Consultez les chapitres d'*Operating Systems: Three Easy Pieces* pour des rappels sur l'API Unix de gestion des processus et la gestion de la mémoire en C.

Vous pouvez consulter les *man pages* de Linux en ligne.

Évaluation

Pour espérer obtenir la note maximal votre travail devra répondre à un certain nombre d'exigences présentées ci-dessous.

Respect des spécifications

- Votre programme devra implémenter les fonctionnalités présentées dans les spécifications et respecter le mode de fonctionnement décrit.

Qualité du code et fiabilité

- Choisissez une convention de nommage pour vos noms de variables et de fonctions et tenez vous-y. Par exemple, utilisez du `snake_case`. Utilisez des noms explicites pour vos variables et fonctions.
- Assurez vous que votre code est formatté. Vous pouvez par exemple utiliser ClangFormat pour cela.
- Votre programme devra être modulaire : répartissez le code dans de multiples fonctions, chacune ayant une responsabilité propre et limitée.
- Assurez vous que votre code compile avec les options `-Wall -Wextra -Werror -std=c17`.
- Votre programme ne devra pas comporter de fuite mémoire : après chaque appel à `malloc`, un appel à `free` doit avoir lieu pour libérer la mémoire allouée. Vous pouvez utiliser Valgrind pour détecter d'éventuelles fuites mémoires.

Tests

- Votre programme devra contenir des **tests unitaires**. Dès lors qu'une fonction est testable unitairement, il recommandé d'inclure un ou plusieurs tests pour vérifier qu'elle a bien le fonctionnement attendu.
- Vous pouvez utiliser un *framework* de tests unitaires pour vous simplifier la vie. MinUnit est un framework minimal qui devrait faire l'affaire, mais vous pourriez tout aussi bien vous débrouiller avec une approche propre.

Documentation

- Dans votre code, incluez des `/* commentaires */` pour donner des indications sur le fonctionnement interne de votre programme. Il est particulièrement judicieux de fournir des commentaires pour documenter les fonctions clés dans votre programmes.
- Vous devez inclure dans votre rendu un fichier `README.md` qui indique comment **compiler** votre programme, comment l'**exécuter** et comment lancer les **tests**.

Originalité et authenticité

- Vous êtes uniquement autorisé à utiliser la bibliothèque standard C. Tout autre bibliothèque externe est proscrite (à l'exception d'un éventuel *framework* de tests).
- Le code produit doit être le vôtre. L'authenticité de votre code sera vérifiée.

Utilisation de Git/GitHub

- Committez régulièrement sur GitHub classroom au fur et à mesure que vous progressez dans le développement de votre programme. Utilisez des messages de commit suffisamment explicites.