Travail pratique #1

IFT-2245

January 29, 2020

Dû le 14 Février à 23h59

Préambule

Attention! Ce TP est le premier d'une suite de TPs qui vont se suivre. Il est donc important pour vous de faire le meilleur TP que vous pouvez faire, et de coder le plus clairement possible, puisque les prochains TPs de la série vont demander de réutiliser le code écrit. À chaque TP, vous aurez l'occasion de prendre une version corrigée. Cependant, vous ne serez peut-être pas à l'aise avec la structure du code, ce qui pourrait vous ralentir. Afin de s'assurer de pouvoir terminer correctement, on conseille de comparer votre solution avec le corrigé.

Introduction

Ce TP vise à vous familiariser avec la programmation système dans un système d'exploitation de style POSIX.
Vous devrez implémenter un terminal/shell typique Linux. Comme tous les autres shell, il devra pouvoir appeler des commandes linux typiques (comme `ls` ou `cat`). Il devra aussi implémenter un sous-ensemble de commandes bash pour agencer le lancement de plusieurs programmes en même temps.
Ce TP sera moins explicite sur le `comment` implémenter les menues fonctions qui permettront de compléter le travail. En effet, c'est ce `comment` qui est la difficulté du travail, pas les petites fonctions. Autrement, c'est le design de votre shell qui est évalué, et non la qualité de votre code.
Attention! Comme c'est une fonctionnalité si standard d'un OS, il existe des fonctions déjà implémentées en C qui feraient essentiellement tout ce TP pour vous. Dès que nous trouverons une telle fonction dans votre remise, vous aurez la note de zéro.

Où coder votre shell?

Le "point d'entrée" de votre shell doit être la fonction `shell()`. Vous ne pouvez vous fier sur `main()`: nous allons complètement le replacer lors de la correction.

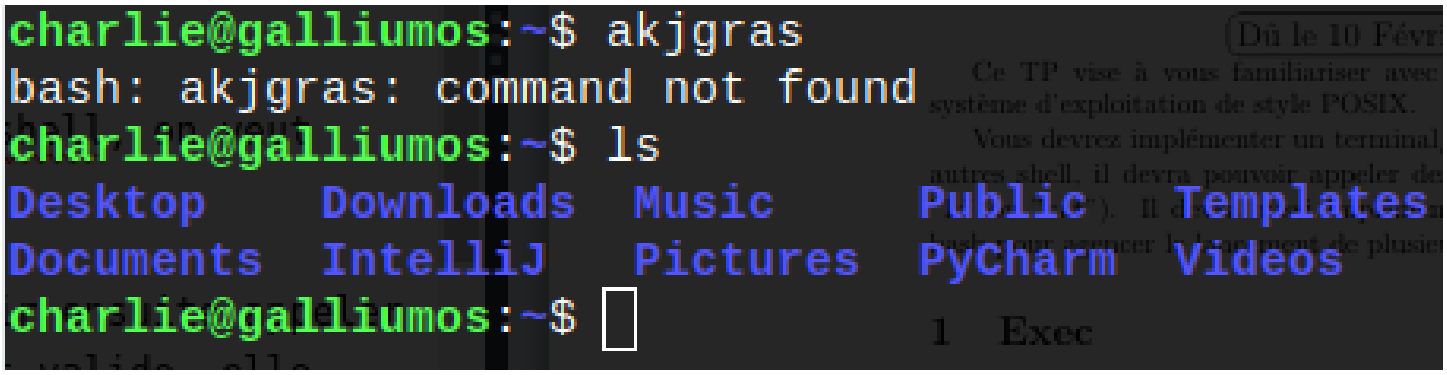
1 Exec

Commençons par l'essentiel. Lorsqu'on lance votre shell, on veut pouvoir exécuter un seul programme.
Donc, lorsqu'on appelle votre shell, on doit pouvoir ensuite appeler `ls` ou `cat` ou `lakjgea`. Lorsque la commande est valide, elle doit imprimer la sortie attendue. Lorsque la commande n'est pas valide, elle doit imprimer `bash: 'la commande': command not found` ('la commande' étant la commande qui a échoué).
Ceci implique bien sûr d'implémenter une ligne de commande qui lit des caractères jusqu'à ce qu'on tape ENTER.

1.1 readLine

Implémentez la fonction readLine, qui lit une ligne entière de la ligne de commande.

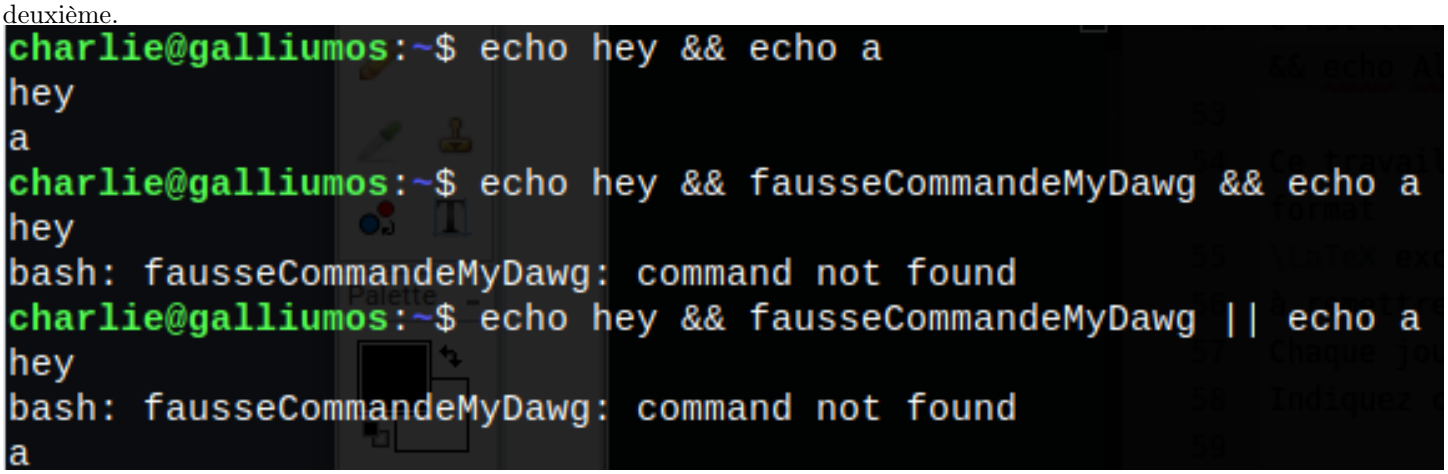
Ensuite, vous devrez utiliser la fonction `execvp` pour exécuter la commande lue par `readLine`. Remarquez que `execvp` est une fonction assez violente: elle écrase le processus en cours. Indice: utilisez un `fork` judicieusement...



Notez que l'exécution séquentielle de plusieurs lignes devrait être supportée. Si l'usager entre `sleep 10` et `echo a` un après l'autre, le shell devrait correctement attendre 10 secondes, puis imprimer "a".

2 ET et OU

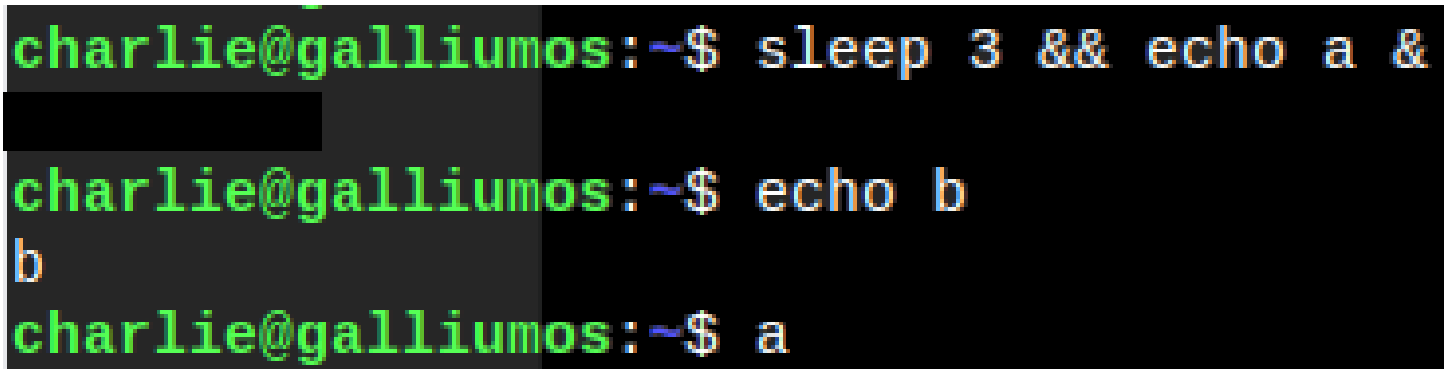
Ensuite, on aimerait que le shell soit capable d'exécuter plusieurs commandes, ou même d'exécuter des commandes conditionnellement. C'est le but des opérateurs && et ||.
&&: Lorsqu'on demande `echo Salut && echo Allo`, le shell devrait imprimer Salut et Allo, sur deux lignes différentes. C'est le cas puisque echo a bien réussi son exécution, donc && passe à la prochaine commande. Cependant, si on demande `CeciNEstPasUneCommandeMyDude && echo allo`, on ne recevrait que l'usuel `bash: CeciNEstPasUneCommandeMyDude: command not found` en sortie.
||: Lorsqu'on demande `echo Salut || echo Allo`, le shell devrait imprimer Salut, et c'est tout. C'est le cas puisque || a vu que la dernière commande a réussi, donc il arrête l'exécution. Si la première commande avait échoué, il aurait passé à la deuxième.



Ici, on ne peut vous donner de sous-question. Bien sûr, vous devrez probablement briser la ligne lue en plus petits blocs. Vous devrez ensuite extraire les symboles propres au shell que vous construisez (en ce moment, seulement "&&" et "||"). Et ensuite écrire une fonction qui chaîne ou arrête l'appel des commandes selon ces symboles. Mais le *comment* est une grande partie de la difficulté de ce TP: vous devrez trouver vous-même.

3 Arrière-plan

Lorsqu'on met `&` à la fin d'une ligne, votre shell doit placer la ligne entière en arrière-plan. Donc, si on appelle `sleep 10 && echo a &` et tout de suite après `echo b`, b devrait imprimer avant a.
Il n'y a pas de limite sur le nombre de lignes mises en arrière-plan. Mais chaque ligne s'exécute toujours séquentiellement (du point de vue de cette ligne-là).
Vous pouvez prendre pour acquis que tous les "&" seront en fin de ligne.
Vous n'avez pas besoin d'écrire le "[1] PID" qui est normalement imprimé dans un vrai shell.



Indice: vous pouvez utiliser `fork` avant l'exécution des lignes appropriées.

4 Commandes fictives

On décide d'ajouter deux fonctions à notre shell: rN et fN.
Ceci change la syntaxe du shell comme suit: `rN(commande X, possiblement avec plusieurs mots) -> exécute la commande X N fois.` `fN(commande, possiblement avec plusieurs mots) -> ne fait rien.`
Donc: `r3(echo a)` appelle trois fois `echo a`. Et `f5(ls)` ne fait rien. Attention! fN n'imprime pas de message d'erreur: à son appel, absolument rien ne se passe.
Vous pouvez voir rN comme étant une suite de && entre N commandes. Cependant, rN ne regarde aucunement si les commandes ont été exécutées avec succès: il les appelles une à la suite de l'autre, sans questions.

Comment tester votre TP

Lorsque vous codez en C, il est extrêmement important de tester votre code. Afin de bien tester votre code, vous avez plusieurs approches.

- Faire des "tests unitaires" en C
- Exécuter manuellement des tests

Ces approches vont bien fonctionner pour tester une fonction C particulière. Cependant, afin de s'assurer que votre TP fonctionne parfaitement, on vous conseille de faire un shell script (.sh) de cette façon:

```
if [ "$(echo "des_commandes" | ./mon_tp)" = "$( des_commandes )$" ]; then
    echo "OK!"
else
    echo "Fail!"
fi;
```

Ce script (en particulier) exécute "des_commandes" sur le fichier "mon_tp" et compare le résultat versus ce que le shell de votre PC fait. Si vous utilisez le WSL, vous devrez exécuter un tel script dans le shell "ubuntu".

Remise

Ce travail est à faire en groupes de 2 étudiants. Le rapport, au format pdf exclusivement et le code sont à remettre par remise électronique avant la date indiquée.
Chaque jour de retard est -15%, mais après le deuxième jour la remise ne sera pas acceptée.
Indiquez clairement votre nom et votre matricule dans un commentaire en bloc au début de chaque fichier, dans ce format:

```
Prenom1 NomDeFamille1 Matricule1
Prenom2 NomDeFamille2 Matricule2
```

Si un étudiant préfère travailler seul, libre à lui, mais l'évaluation de son travail n'en tiendra pas compte. Nous n'aiderons pas ceux qui ne se trouvent pas de partenaire à s'en trouver un.
Le programme doit être exécutable sur les ordinateurs du DIRO.
Pour la remise, vous devez remettre deux fichiers (main.c et rapport.pdf) en utilisant **ce lien** dans un fichier zip. La valeur totale de ce TP est 10 points. Assurez-vous que tout fonctionne correctement sur les ordinateurs du DIRO.

Barème

- Votre note sera divisée équitablement entre chaque question et un test global qui vérifie si tout fonctionne bien ensemble. Donc, avec 4 questions et un test global, on obtient 20.00% par question et pour le test global.
- Tout usage de matériel (code ou texte) emprunté à quelqu'un d'autre (trouvé sur le web, etc.) doit être dûment mentionné, sans quoi cela sera considéré comme du plagiat. Si pour une question votre solution est directement copiée, même si il y a attribution de la source, cette question se verra attribuée la note de zéro. Vous pourrez cependant l'utiliser dans les sections suivantes sans pénalité.
- Vous serez pénalisés pour chaque warning lors de la compilation à raison de 1% par warning, sauf pour les warning reliés à l'assignation à NULL, à la comparaison avec NULL, et aux override des fonctions de librairie.
- Si une fuite mémoire mémoire est identifiée, vous perdrez 15%. Vous ne perdrez pas plus de points pour les fuites si vous en avez une ou trente.
- Les accès mémoire illégaux identifiés par Valgrind entraîneront jusqu'à 5% de perte, à raison de 1% par accès. La répétition d'un même accès sera comptée comme 1% de plus quand même.
- Votre devoir sera corrigé automatiquement en très grande partie. Si vous déviez de ce qui est demandé en output, les points que vous perdrez seront perdus pour de bon. Si vous n'êtes pas certains d'un caractère demandé, demandez sur le forum studium et nous répondrons de façon à ce que chaque étudiant puisse voir la réponse.
- La méthode de développement recommandée est d'utiliser CLion et son intégration avec Valgrind. Si vous voulez utiliser d'autres techniques, vous pouvez le faire, mais nous ne vous aiderons si vous rencontrez des problèmes avec ces techniques.