Langage C et Programmation Système
TP n° 11 : Implémentation d'un shell – Partie 2

Suite du TP n° 10 : Le but de ce TP est de compléter ce qui a été fait dans le TP précédent de façon à ce que notre shell simplifié `mysh` puisse gérer les redirections ainsi que les tubes anonymes entre processus. Il vous est donc demandé de repartir du travail réalisé dans le TP précédent. (Vous pouvez aussi vous servir du corrigé disponible sur DidEL.) Si vous avez implémenté la commande interne `jobs` (exercice optionnel du TP n° 10), ignorez-la pour le moment. Le vrai job control est prévu pour le prochain TP.

Exercice 1 : Redirections

Le but de cet exercice est de faire en sorte que notre shell gère les redirections des entrées-sorties standards vers des fichiers.

1. Avant toute chose, vous devez vous familiariser avec la façon de faire des redirections en C. Analysez ce que font les deux programmes suivants, en les testant et en vous aidant des pages de manuel :

```
#include <stdio.h>
#include <fcntl.h>
#include <unistd.h>

int main(int argc, char *argv[]) {
    int f;
    if (argc != 3) {
        fprintf(stderr, "Usage: %s FILE TEXT\n", argv[0]);
        return 1;
    }
    if ((f = open(argv[1], O_WRONLY|O_APPEND|O_CREAT,
                  S_IRUSR|S_IWUSR)) == -1)
        return 1;
    if (dup2(f, STDOUT_FILENO) == -1)
        return 1;
    if (execlp("echo", "echo", argv[2], NULL) == -1)
        return 1;
}
```

```
#include <stdio.h>
#include <fcntl.h>
#include <unistd.h>

int main(int argc, char *argv[]) {
    int f;
```

```
    if(argc != 2) {
        fprintf(stderr, "Usage: %s FILE\n", argv[0]);
        return 1;
    }
    if ((f = open(argv[1], O_RDONLY)) == -1)
        return 1;
    if (dup2(f, STDIN_FILENO) == -1)
        return 1;
    if (execlp("cat", "cat", NULL) == -1)
        return 1;
}
```

2. Vous allez commencer par gérer les redirections de l'entrée standard. Dans notre shell, une commande de la forme `< file cmd` indiquera que l'on souhaite rediriger l'entrée standard de `cmd` vers le fichier `file`. **Nous n'autoriserons pas dans ce TP les redirections pour les commandes *built-in***. Dans le fichier `mysh.h`, vous ajouterez la signature de la fonction `int execute_command_redirect(int argc, char **argv)`. Le code de cette fonction sera à ajouter dans un nouveau fichier `redirect.c`. La fonction `execute_command_redirect` fonctionne à peu près de la même façon que la fonction `execute_command_external`, mis à part que dans le processus fils exécutant la commande, si elle commence par `<` on doit d'abord rediriger l'entrée standard vers le fichier fourni. N'oubliez pas de gérer le cas où la commande est exécutée en arrière-plan. Cette commande sera ainsi appelée depuis la fonction `execute_command`.
3. Modifiez maintenant votre code (fonction `execute_command` et `execute_command_redirect`) pour que votre système puisse gérer les commandes de la forme `> file cmd`. On supposera pour cette question qu'il n'y a pas de redirection dans `cmd`. Nous vous conseillons pour faire un code compact, dans la fonction `command_redirect` d'utiliser des variables vous permettant de stocker les différentes redirections et d'éviter la duplication de code (par exemple, vous pouvez avoir un entier pour vous rappeler si il faut faire ou non une redirection de l'entrée standard et de la même façon pour la sortie standard, et au moment d'effectuer ces redirections vous testez les valeurs de ces entiers).
4. Finalement, procédez de la même façon pour gérer la redirection de la sortie d'erreur avec des commandes de la forme, `>2 file cmd`.
5. Modifiez maintenant votre fonction de façon à gérer les redirections multiples, c'est-à-dire les commandes de la forme `< file > file2 >2 file3 cmd`.
6. Testez que vos différentes redirections fonctionnent correctement. En ce qui concerne la redirection des erreurs, comment procédez-vous ?

Exercice 2 : Tubes anonymes

Dans cet exercice, nous allons faire en sorte que notre shell `mysh` gère les tubes anonymes, c'est-à-dire les commandes de la forme `cmd1 | cmd2 | ... |cmdn`. Dans ce TP, nous supposerons que les commandes ainsi chaînées sont toutes externes. Dans un premier temps, nous supposerons également qu'aucune de ces commandes ne fait de redirections. Ainsi, lorsque la fonction `execute_command_externa`l sera appelée, il faudra tester si la commande passée en arguments est une commande avec tubes anonymes ou non.

1. Récupérez sur DidEL ou dans `/ens/sangnier/sysc/tp11/` les fichiers `piped.h` et `piped.h` qui vous fournissent une fonction

   ```
   int separate_piped_proc(int argc, char **argv, pipedcmd *pc);
   ```

   permettant d'obtenir le nombre de processus chaînés et mettant ces commandes chaînées dans le tableau `pc` (consultez la définition du type de structure `pipedcmd` dans `piped.h`).

2. Complétez maintenant la fonction `execute_command_external` pour gérer le cas avec des tubes anonymes. Les lignes suivantes indiquent comment vous pourrez vous servir de `separate_piped_proc` (pensez à inclure le fichier `piped.h`) :

   ```
   int execute_command_external(int argc, char** argv) {
       pipedcmd pcmd[MAXPIPED];
       int np = separate_piped_proc(argc, argv, pcmd);
       if (np == 1) {
           /* Votre ancien code pour 'execute_command_external'. */
       } else {
           /* Ici, le cas ou l'on a au moins un tube anonyme. */
       }
   }
   ```

   Votre fonction pourra fonctionner de la façon suivante. Dans le cas d'une commande de la forme `cmd1 | cmd2 | ... |cmdn`, le processus père créera un fils qui gérera la commande `cmdn`, et il attendra la terminaison de ce fils (on testera que `cmdn` ne se termine pas par `&`, ce cas sera traité ensuite). Le processus fils gérant `cmdn` créera les $(n-1)$ processus gérant les commandes `cmd1`, `cmd2`, ..., `cmd(n-1)` et les tubes servant à la communication. Il faudra ensuite faire les bonnes redirections des entrées-sorties standards des fils vers les tubes adéquats. Attention à ne pas rediriger l'entrée standard du processus gérant la commande `cmd1`. On rappelle que lorsqu'un tube anonyme est placé entre deux commandes, alors la sortie standard **et** la sortie d'erreur de la première commande sont redirigées vers l'entrée standard de la deuxième commande.

3. On souhaite maintenant autoriser ces tâches à être lancées en arrière-plan (en mettant un `&` au bout de la dernière commande). Faites en sorte que votre shell puisse gérer ce cas.

4. On souhaite autoriser les redirections de la forme `<` dans la première commande, et de la forme `>` dans la dernière commande. Modifiez votre code de façon à ce que cela soit possible.

5. *Optionnel :* Vous pouvez proposer à l'utilisateur de votre shell une autre façon de gérer les commandes de la forme `cmd1 | cmd2 | ... |cmdn`, en faisant en sorte que le processus gérant la commande `cmdi` soit fils du processus gérant la commande `cmd(i+1)`. Le choix du mode de fonctionnement pourrait se faire via un mot-clé du shell. Par exemple, `pipelast` garantirait que l'enchaînement de processus soit fait comme dans la question 2, et `pipenext` ferait en sorte que les processus soient chaînés de la manière que l'on vient de le décrire. Par défaut, on prendrait le mode `pipelast`. Il faudrait alors stocker le mode choisi dans une variable globale.