Langage C et Programmation Système
TP n° 10 : Implémentation d'un shell – Partie 1

Le but des TPs n° 10, 11 et 12 : Les trois dernières semaines de TP sont consacrées à l'implémentation d'un shell Unix simplifié que nous appellerons `mysh`. Quand on saisira `./mysh` dans le terminal, le shell actuel (probablement `bash`) créera un processus enfant exécutant `mysh`, et lui confiera la responsabilité d'interagir avec l'utilisateur. Après cela, les commandes entrées par l'utilisateur seront interprétées par `mysh`, jusqu'à ce que la session soit terminée avec la commande spéciale `exit` (qui rend la main au shell parent).

Exercice 1 : Préparation

Vous ne commencez pas à zéro, on vous fournit un squelette de code.

1. Téléchargez et décompressez l'archive `mysh.tar.gz` dans votre répertoire de travail. (Vous la trouverez sur DidEL ou localement sous `/ens/reiter/sysc/tp10`.)
2. Compilez le programme (via `make`), exécutez-le (`./mysh`), et jouez un peu avec.
   Ce n'est pas encore très intéressant, parce que le shell ne comprend qu'une seule commande : `exit` (que l'on peut aussi entrer à travers le raccourci clavier Ctrl+D). Par défaut, `exit` termine le processus `mysh` en renvoyant la valeur `0`, mais vous pouvez aussi spécifier une autre valeur non négative en argument (par exemple `exit 37`).
   Comment récupérer cette valeur ? Cela dépend du shell parent. Si vous êtes sous `bash`, vous pouvez utiliser `echo $?` pour afficher le code de retour de la dernière commande exécutée (en l'occurrence `mysh`). L'équivalent sous `fish`[1] est `echo $status`, et sous `mysh`, quand vous l'aurez implémenté, ce sera `status` (une commande interne).
3. Jetez un coup d'œil sur le `Makefile`. Il est un peu plus générique que celui vu en TP n° 3. Cela pourrait vous être utile pour vos futurs projets.
4. Familiarisez-vous avec le code déjà existant dans `mysh.h`, `main.c` et `builtin.c`. Ce n'est pas long, mais il est important que vous compreniez la structure du code avant de passer à la suite. La boucle infinie suivante (cf. `main.c`) est l'élément central du shell :

```
while (1) {
    display_prompt();
    read_command(argl);
    argc = tokenize_command(argl, argv);
    status = execute_command(argc, argv);
}
```

   Dans chaque itération,
   - on affiche l'invite de commande `"mysh$ "` (ou une autre chaîne de votre choix),
   - on lit sur `stdin` la prochaine ligne de commande `argl` entrée par l'utilisateur,
   - on découpe cette ligne pour obtenir un vecteur (tableau) d'arguments `argv`,
   - on essaye d'exécuter la commande souhaitée, et on stocke sa valeur de retour.

   Dans la dernière étape, les variables `argc` et `argv` sont utilisées de la même manière que dans une fonction `main` comportant de tels arguments.

1. Publicité : **fish** – the **f**riendly **i**nteractive **sh**ell `http://fishshell.com`

Exercice 2 : Commandes externes (`external.c`)

Le rôle principal d'un shell est de permettre à l'utilisateur d'exécuter les programmes installés sur le système, qui sont stockés sous forme de fichiers exécutables indépendants. Avec les connaissances que vous avez acquises dans le TP n° 9, cette fonctionnalité est facile à ajouter à votre shell, et elle le rendra immédiatement plus intéressant.

1. Dans le fichier `external.c`, complétez la fonction `execute_command_external` qui essaye de lancer la commande donnée par `argv`. Elle procède comme suit : Le shell crée un processus enfant (cf. `fork(2)`), et attend que celui-ci termine son exécution (cf. `waitpid(2)`). Si tout se passe bien, la fonction `execute_command_external` du processus parent retourne le code de retour de l'enfant. Sinon, elle affiche un message d'erreur utile sur `stderr` (indiquant le problème), et retourne `1`. L'enfant, quand à lui, essaye d'exécuter la commande spécifiée par `argv` (cf. `execvp(3)`). En cas de succès, le code exécuté par l'enfant est remplacé par celui de la commande appelée (qui prend donc contrôle du processus enfant), et il n'y a plus rien à faire (on ne retournera plus au code original). Sinon, l'enfant doit afficher un message d'erreur sur `stderr`, et *terminer* (pas simplement retourner de la fonction) avec le code de retour `1` (cf. `exit(3)`).

   Afin de pouvoir utiliser la fonction que vous venez d'écrire, vous devez l'appeler depuis la fonction `execute_command` dans `main.c`.
2. Maintenant, essayez de lancer quelques programmes depuis `mysh`. Tout est permis, vous pouvez même exécuter `mysh` depuis une instance de `bash` qui, elle-même, a été appelée depuis une autre instance de `mysh`, etc. ; cela ne devrait poser aucun problème.

   Lancez aussi un programme avec une interface graphique (par exemple `xterm`, s'il est disponible sur votre système). Durant l'exécution de ce programme, le processus `mysh` ne devrait pas répondre à vos commandes, puisqu'il attend que le processus enfant termine. (Les commandes entrées sont interprétées a posteriori.)
3. Ajoutez à votre shell la possibilité de lancer un programme en arrière-plan, quand on écrit le symbole `&` à la fin d'une ligne de commande. Pour nos besoins, il suffit qu'en mode arrière-plan, dans la fonction `execute_command_external`, après l'exécution de `fork`, le parent affiche l'identifiant de processus de l'enfant et retourne immédiatement avec la valeur `0`, sans attendre la terminaison de l'enfant.

   Testez cette nouvelle fonctionnalité avec `xterm &` (ou un autre programme ayant une interface graphique). Vous devriez pouvoir utiliser `mysh` et `xterm` en parallèle. Que se passe-t-il si vous terminez `mysh` ?

Exercice 3 : Commandes internes (`builtin.c`)

Bien que votre shell soit désormais capable d'exécuter n'importe quel programme sur le système, vous ne pouvez toujours pas changer de répertoire courant (pas de `cd`). La raison est que, sous Unix, chaque processus peut définir son répertoire de travail indépendamment des autres. Si le shell créait un nouveau processus pour `cd`, alors ce dernier aurait accès à son propre répertoire courant, mais pas à celui du shell. La fonctionnalité de `cd` doit donc être directement intégrée dans le shell.

Ce type de commandes, qui ne correspondent pas à des exécutables autonomes, sont appelées des commandes internes (« builtins »). Pour des raisons d'efficacité, la plupart des shells offrent également des commandes internes pour certaines tâches simples qui pourraient tout aussi bien être effectuées par un autre programme. (Cela évite la création d'un nouveau processus.)

1. Dans votre shell de travail (`bash`, `fish`, `zsh`, ...), utilisez la commande `type` pour identifier les « builtins » parmi les commandes que vous avez l'habitude d'utiliser. Qu'en est-il de `cd`, `pwd`, `ls`, `echo`, `cat`, `cp`, `grep`, `less`, `man` et `exit` ?
2. Ajoutez à votre shell `mysh` la commande interne `status` qui affiche la valeur retournée par la dernière commande exécutée. Elle ne prend pas d'argument.

   (Pour cette commande, ainsi que pour celles des questions suivantes, mettez le code dans la fonction `execute_command_`*`commande`*, qui se trouve dans `builtin.c`, et n'oubliez pas d'ajouter l'appel correspondant dans `execute_command`.)
3. Implémentez la commande `cd` qui change le répertoire courant du shell par l'intermédiaire de l'appel système `chdir(2)`. Si le chemin n'est pas spécifié, `cd` doit aller dans le répertoire personnel de l'utilisateur, dont le chemin est stocké dans la variable d'environnement `HOME` (cf. `getenv(3)`). En cas d'échec, la commande doit afficher un message utile sur `stderr` (répertoire inexistant, pas de droit d'accès, etc. ; cf. `access(2)`).

   Optionnel : Il peut aussi être agréable de pouvoir facilement retourner au répertoire précédent avec la commande `cd -`. Pour cela, vous pouvez tenir à jour les variables d'environnement `PWD`, contenant le chemin *absolu* du répertoire courant, et `OLDPWD`, contenant celui du répertoire précédent (cf. `setenv(3)` et `getcwd(3)`).
4. Ajoutez la commande interne `pwd` qui se comporte exactement comme la commande externe du même nom, déjà présente sur le système. Si vous avez implémenté la commande `cd -`, comme suggéré dans la question précédente, il suffit d'afficher la valeur de la variable d'environnement `PWD`.
5. Vérifiez, à l'aide de `status`, que les commandes internes `exit`, `status`, `cd` et `pwd` de votre shell `mysh` retournent bien `0` en cas de succès et `1` en cas d'échec.

   Vérifiez aussi que vous voyez bien le code de retour des commandes externes. Pour cela, lancez un deuxième processus $\texttt{mysh}_2$ depuis votre processus actuel $\texttt{mysh}_1$, et exécutez `exit` i dans $\texttt{mysh}_2$, avec une valeur i différente de `0` et `1`. Si `status` dans $\texttt{mysh}_1$ affiche une autre valeur que i, alors vous avez probablement une petite erreur dans votre fonction `execute_command_external`.

Exercice 4 : Un avant-goût de job control (*Optionnel*) (`job.c`)

Il est fort probable que l'on reviendra plus en détail sur la notion de job (tâche) dans le TP n° 12. Mais, en attendant, vous pouvez déjà implémenter une une commande interne `jobs` qui affiche la liste de tous les processus en arrière-plan (lancés avec le symbole `&` en fin de commande).

1. Crééz un nouveau fichier `job.c` et définissez-y les variables globales suivantes :

   ```
   static int jobcount = 0;
   static pid_t jobtab[MAXJOB];
   ```

   Le compteur `jobcount` donne le nombre de processus en arrière-plan, et le tableau `jobtab` stocke les identifiants de ces processus (PIDs). L'identifiant de job (JID) d'un processus est simplement son indice dans le tableau `jobtab`. On considère qu'une case dans `jobtab` est libre si sa valeur est `0`.

   La macro `MAXJOB` est à définir dans `mysh.h` (à vous de choisir une valeur).
2. Écrivez les fonctions suivantes dans `job.c`, et mettez les signatures correspondantes dans `mysh.h`.

`int job_table_full();`
Renvoie une valeur non nulle si et seulement si `jobtab` est plein.

`int add_job(pid_t pid);`
Stocke `pid` dans la première case libre de `jobtab` et renvoie l'indice de cette case. Incrémente `jobcount`.

`void display_running_jobs();`
Affiche sur `stdout` la liste des jobs en cours d'exécution.
(Pour chaque job, elle affiche une ligne comportant le JID et le PID associé.)

`void refresh_job_table();`
Supprime du tableau `jobtab` les processus qui ont terminé leur exécution. Évidemment, pour chaque case vidée, il faut décrémenter `jobcount`. Afin de déterminer si un processus est encore en cours d'exécution, vous pouvez vous servir de l'appel système `waitpid(2)` avec l'option `WNOHANG`. Pour chaque processus terminé, la fonction affichera un message sur `stdout`, indiquant le JID, le PID et la valeur retournée par le processus.

3. On veut que `jobtab` soit à jour quand le shell affiche la liste des jobs ou quand il essaye de lancer un nouveau processus en arrière-plan. Par ailleurs, il est désirable que l'utilisateur soit informé rapidement quand des processus ont terminé leur exécution. Une manière simple de garantir cela est d'appeler `refresh_job_table` dans la boucle principale de `mysh` (cf. `main.c`). Insérez cet appel à un endroit judicieusement choisi.

4. Il faut légèrement adapter la fonction `execute_command_external` pour qu'elle soit compatible avec notre notion de job (rudimentaire) : Si `jobtab` est plein, le shell refuse de lancer un autre processus en arrière-plan. De plus, si un processus est exécuté en arrière-plan, il est ajouté à `jobtab`, et le shell affiche son JID, en plus du PID qui était déjà affiché dans la version précédente. (Servez-vous des fonctions de la question 2.)

5. Pour finir, ajoutez à votre shell la commande interne `jobs`, qui affiche les jobs en cours d'exécution (cf. `main.c` et `builtin.c`).

Cette commande est utile en combinaison avec `kill`, qui permet de terminer un processus dont on connaît le PID. La commande `kill` est disponible sur le système comme exécutable indépendant, mais de nombreux shells la fournissent aussi comme commande interne ... ☺

Félicitations ! Vous venez d'implémenter un mini-shell Unix (encore limité, certes, mais néanmoins fonctionnel). *N'hésitez pas à être créatif et à écrire vos propres extensions !*

Ce qui vous attend :
Durant les deux semaines restantes, vous aurez l'occasion d'ajouter des fonctionnalités à votre `mysh`. Voici le plan préliminaire. (Sans garantie, les sujets n'étant pas encore rédigés.)

- **TP n° 11 :**
  - Rediriger les entrées/sorties d'un processus. (`<`, `>`, `>>`, `2>`, ...)
  - Chaîner des processus par des tubes. (`prog1 | prog2 | prog3`)
- **TP n° 12 :**
  - Job control : Gérer plusieurs processus ou groupes de processus. (`jobs`, `bg`, `fg`)
  - Envoyer des signaux à des processus en cours d'exécution. (`kill`)