Université Joseph Fourier, Grenoble
Licence d'Informatique L3

Introduction aux Systèmes et Réseaux

TP n°4 : Réalisation d'un mini-*shell*

Ce TP (qui sera prolongé par une Apnée) consiste à réaliser un mini-*shell* pour Unix en partant d'un *shell* très simple existant [1]. Les principes de cette extension ont été donnés dans le TD4 et sont rappelés ci-après. Le programme du *shell* existant est dans le placard du TP4, en 3 morceaux comme vu en TD : le programme principal `myshell.c`, le programme d'interprétation des commandes `eval.c` et l'analyseur de lignes de commande `parseline.c`. Il faut y ajouter les fichiers d'inclusion `csapp.h` et `myshell.h` ainsi que la bibliothèque `csapp.c`.

On commencera par faire fonctionner ce *shell* (lancé au-dessus du *shell* Unix courant). Noter que les commandes doivent donner le nom complet des fichiers exécutables (partant de la racine) puisque le *shell* ne recherche pas le fichier exécutable dans le `PATH`.

1 Travail à réaliser

On reprend ce qui a été vu au cours du TD. Le but du mini-projet est de construire un *shell* plus complet, partant des programmes ci-dessus. Ce *shell* doit avoir les caractéristiques suivantes (analogues à celles des *shells* Unix courants) :

- Les commandes intégrées sont exécutées par le processus *shell* lui-même, les autres par un fils. Si la commande se termine par `&`, elle est exécutée en arrière-plan, sinon en premier plan.
- Les commandes exécutées en arrière-plan s'appellent des *jobs* [2]. Un *job* peut être désigné par le `PID` du processus qui l'exécute (exemple 14567) ou par son numéro de *job* précédé de `%` (exemple `%3`). Les numéros de *jobs* sont des entiers positifs, attribués à partir de 1.
- La frappe de *control-c* et *control-z* doivent respectivement envoyer un signal `SIGINT` et un signal `SIGTSTP` au(x) processus de premier plan (voir détails plus loin).
- La commande intégrée `jobs` doit lister tous les *jobs* avec leur état, le texte de la commande qu'il exécutent et le `PID` du processus qui les exécute. Exemple :

```
<myshell> jobs
[1] 3456  Stopped   myprog param1 param2
[2] 3460  Running   /usr/local/bin/nedit prog.c
<myshell>
```

- Les commandes intégrées `fg`, `bg` et `stop` s'appliquent à un *job*, désigné soit par son numéro de *job* soit par son numéro de processus. Les effets de ces commandes sont résumés sur la figure vue dans le TD n°4 (et le cours n°2), qui rappelle les états des travaux et les transitions entre ces états.

1. emprunté (avec des adaptations) à l'ouvrage : R. E. Bryant, D. O'Hallaron. *Computer Systems: a Programmer's Perspective*, Prentice Hall, 2003.

2. Par abus de langage, on utilisera le terme *job de premier plan* pour désigner une tâche en cours d'exécution au premier plan (soit parce qu'elle a été lancée directement en premier plan, soit parce qu'elle a été positionnée en premier plan ultérieurement).

Noter qu'un processus lancé en premier plan ou en arrière plan peut créer des processus fils. Par défaut, ceux-ci forment avec leur père un groupe, dont le numéro *pgid* est le *pid* du père. Les commandes (`bg`, `fg`, `stop`) et les signaux générés par *control-c* et *control-z* s'adressent en fait à tous les processus du groupe de premier plan.
- Le *shell* doit ramasser tous ses fils zombis (ce que ne fait pas la version initiale ; le vérifier en exécutant un programme en arrière-plan et en faisant `/bin/ps` après la terminaison de ce programme).
- Si un *job* se termine parce qu'il reçoit un signal qu'il ne traite pas, le *shell* doit imprimer un message indiquant le numéro du *job* et le numéro du signal.
- La commande intégrée `wait` permet d'attendre la fin (ou la suspension) de tous les *jobs* en cours. Avant de rendre la main, elle affiche une ligne décrivant l'état (terminé ou suspendu) de chacun des *jobs* qu'elle a attendus.
- La terminaison ou la suspension d'un *job* d'arrière plan doit déclencher un affichage pour informer l'utilisateur (numéro de *job* et type d'événement). Cependant, afin de ne pas perturber l'utilisateur (qui peut être en train de saisir une nouvelle commande ou d'utiliser une tâche de premier plan), l'affichage du message doit être effectué de manière différée, juste avant l'affichage d'un nouvel invite de commande.

2 Éléments d'analyse

On reprend quelques éléments d'analyse dont certains ont été vus lors du TD n°4.

2.1 Structures de données

Un travail (*job*) sera représenté par une structure contenant les éléments suivants : numéro du *job* (entier > 0), *pid* du processus qui l'exécute, état du *job* (cf. figure), ligne de commande exécutée par le *job* ainsi que toute autre information jugée utile.

Pour simplifier la programmation, il est conseillé d'utiliser un tableau pour contenir ces structures, plutôt qu'une liste chaînée. La taille de ce tableau (`MAXJOBS`) est un paramètre du système (le prendre de l'ordre de 10).

Programmer des fonctions d'accès pour donner notamment le numéro de la première case libre du tableau, le numéro du *job* de premier plan (s'il existe), ajouter un nouveau *job*, supprimer un *job* et libérer la case correspondante du tableau, initialiser le tableau.

2.2 Gestion des travaux

La gestion des *jobs* est le travail principal demandé.

Comme dans les *shells* usuels, `fg` envoie un signal `SIGCONT` au *job* et le fait exécuter au premier plan. La commande `bg` envoie un signal `SIGCONT` au *job* et le fait exécuter en arrière-plan. Noter que :

1. La différence entre un *job* d'arrière-plan et le *job* de premier plan est que la fin de ce dernier (ou plutôt sa disparition du premier plan) est attendue par son père (le processus qui exécute le *shell*). Il faut ici considérer deux points :
   - Il y a plusieurs manières pour le processus de premier plan de cesser d'être au premier plan : suspension (suite à la réception d'un signal `SIGSTOP` ou éventuellement

SIGTSTP), mort par `exit`, mort causée par un traitant de signal. Dans tous ces cas, le système d'exploitation envoie un signal `SIGCHLD` à son père (le processus qui exécute le *shell*).
   - Comment faire attendre le processus *shell* après le lancement d'un processus de premier plan ? Une solution est de le mettre en attente, mais en testant périodiquement, par mesure de sécurité, l'état du processus de premier plan (boucle de test contenant un `sleep(1)`) [3].
2. Les processus d'arrière plan qui se terminent doivent être "ramassés" (c'est-à-dire doivent être traités par `waitpid` pour cesser d'être zombis). Cela doit se faire dans le traitant du signal `SIGCHLD`, cf. détails en 2.3.
3. Il se pose un problème d'exclusion mutuelle : pendant que le *shell* modifie les structures représentant l'état des travaux, il est prudent de masquer certains signaux (dont les traitants peuvent aussi manipuler ces structures), pour préserver la cohérence des données.
4. Un programme lancé par un *shell* fait l'hypothèse que son masque de signaux initial est vide et que ses traitants de signaux correspondent aux traitants par défaut. S'assurer que l'implémentation du mini-*shell* respecte cette règle.

2.3 Quelques points techniques

On indique ici quelques compléments utiles à la réalisation.

Compléments sur `waitpid` Noter que le traitant de `SIGCHLD` du *shell* doit "ramasser" les processus terminés (zombis) et traiter les processus suspendus (changer leur état dans les structures de données du shell), mais ne doit *pas* attendre la fin des processus en cours (travaux d'arrière plan non terminés). Pour cela, on utilisera les options suivantes de la primitive `waitpid` : `waitpid(-1, &status, WNOHANG|WUNTRACED)`

Si la valeur de retour de l'appel est zéro, il n'y a pas/plus de fils suspendu ou terminé à traiter. Sinon si la valeur de retour est strictement positive, la valeur de retour correspond au pid du fils qui a généré le signal. Dans ce cas, on peut connaître la cause de l'envoi du signal en testant la variable `status` (initialisée par `waitpid`) avec `WIFSTOPPED`, `WIFSIGNALED` et `WIFEXITED`. Voir `man waitpid` et les TD/TP précédents pour les détails.

Gestion des groupes En fait, aussi bien le processus de premier plan que les jobs peuvent créer des processus fils. Le *shell* va donc gérer des groupes plutôt que des processus isolés. Rappelons qu'un groupe est constitué d'un processus leader (soit p) et éventuellement d'autres processus (par défaut, les descendants de p). Il porte un numéro de groupe *pgid* qui est égal au *pid* du processus leader p. Par défaut, un processus créé par le processus *mini-shell* appartient au même groupe que ce processus *mini-shell*. On souhaite au contraire créer un nouveau groupe dont ce nouveau processus soit le leader (conformément aux *shells* usuels). Pour cela, il faut que le processus créé exécute la primitive : `setpgid(0, 0)` (voir `man setpgid`).

3. Cette solution a le mérite de simplifier l'implémentation du *shell* car `waitpid` n'est appelée qu'à un seul endroit dans le code (au sein du traitant du signal `SIGCHLD`). Dans le cas contraire, il y a concurrence entre les appels à `waitpid` effectués par le *shell* et ceux effectués par le traitant du signal `SIGCHLD`.

Blocage et déblocage des signaux Se référer aux exercices des TP précédents pour l'utilisation de `sigprocmask` et des primitives de manipulation du type opaque `sigset_t`. Bien noter qu'à l'issue d'un `fork()`, un processus fils hérite de l'état courant du masque de signaux de son père.

Programmes à utiliser pour tester le *shell* Pour tester le bon fonctionnement du *shell*, considérer uniquement des programmes simples et non interactifs (c'est-à-dire qui ne nécessitent pas d'entrées au clavier de la part de l'utilisateur). Des commandes existantes telles que `ps`, `ls` et `echo` correspondent à cette définition. Par opposition, des programmes tels que `more`, `less`, `vim`, `emacs`, `top` modifient la configuration par défaut du terminal et nécessitent un support adapté dans le *shell*, qui ne sera pas étudié dans le cadre de ce TP [4]. Pour les mêmes raisons, il n'est pas possible de faire pleinement fonctionner un *shell* (classique ou mini-*shell*) lancé à partir du mini-*shell* tel qu'il vous est fourni. Des pistes de travail (facultatives) proposées en section 4 expliquent la cause de cette restriction et comment la lever.

3 Présentation des résultats

Chaque binôme rendra le code source *commenté* des programmes réalisés et un bref compte-rendu présentant : (a) les principales modifications introduites dans la structure du *shell*, (b) un résumé du principe des solutions adoptées pour le contrôle des travaux et la gestion des signaux ainsi (c) qu'une description des tests effectués.

Une démonstration des réalisations sera organisée en fin de semestre (détails indiqués en temps utile).

4 Travail facultatif

Plusieurs pistes d'extensions facultatives au *shell* spécifié dans les sections précédentes seront proposées dans le placard électronique. Dans tous les cas, il est demandé de fournir une version complète et fonctionnelle du *shell* de base (donc penser à valider et sauvegarder le code du *shell* de base avant de commencer à introduire des modifications liées aux extensions).

4. Plus précisément, les programmes interactifs qui lisent sur l'entrée du terminal sont à éviter. En revanche, les programmes interactifs associées à une fenêtre graphique, tels que `nedit` ou `xterm`, peuvent être utilisés.