Pablo Rauzy <pr@up8.edu>
https://pablo.rauzy.name/teaching/sr

Systèmes et réseaux

TP 3 : Coquillette

Le but de ce TP est de créer un mini shell basique.
Pensez à consulter les pages de manuel et à compiler votre code régulièrement !

Exercice 0.

Invite de commande.

1. On commence par le *prompt*. On souhaite que celui-ci affiche le nom de l'utilisateurice connecté·e (fonction `getlogin`), le nom du système (fonction `gethostname`), et le répertoire de travail (fonction `getcwd`).
   Par exemple "`pablo@bocal:/home/pablo/sr/tp3$ `".
   → Commencez par écrire un programme qui affiche ces informations sur sa sortie d'erreur standard puis quitte, de sorte à vous assurer que cela fonctionne bien.
2. Un shell est ce qu'on appelle un REPL (*read eval print loop*), c'est-à-dire que sa fonction principale est une boucle qui lit une commande sur son entrée standard, l'exécute, affiche éventuellement ses résultats sur sa sortie standard, puis recommence ; jusqu'à atteindre la fin de son entrée standard (`EOF`).
   → Écrivez une première version de la boucle principale du shell qui :
   — affiche notre prompt,
   — lit une ligne sur son entrée standard (dont il n'est rien fait pour l'instant).
   On sort de la boucle quand on atteint `EOF`.

Exercice 1.

Commandes intégrées.

1. On peut s'inspirer de l'architecture de la boîte à outils du TP 1 pour ajouter un système de commandes intégrées (*builtins*).
   On va dans un premier temps n'avoir qu'une seule commande intégrée : `exit`.
   → Écrivez la fonction `builtin_exit`, dont le prototype est identique à celui de la fonction principale d'un programme.
2. → Déclarez une structure de données `builtin` permettant de stocker un nom et une fonction ayant la même signature que la fonction principale d'un programme.
3. → Déclarez en variable globale constante un tableau de `struct builtin` nommé `builtins`, de longueur 1 (cette devrait être une macro `NB_BUILTINS` par exemple) et initialisée avec le nom et la fonction de notre commande intégrée.
4. → Écrivez une fonction `find_builtin` qui, étant donnée une chaîne de caractère, renvoie l'entier qui est l'indice de la commande intégrée correspondante dans le tableau `builtins` ou `-1` si la commande n'existe pas.
5. Il faut maintenant qu'on puisse exécuter notre commande intégrée.
   → Dans la boucle principale, faites appel à la fonction `find_builtin` pour vérifier si la commande correspond à une commande intégrée, et si c'est le cas, appelez là.
6. → Que fait le code suivant ?

```c
struct command {
  int argc;
  char **argv;
};

int count_args (const char *s)
{
  int n = 0;
  char p = ' ';
  while (*s) {
    if (isspace(p) && !isspace(*s)) n++;
    p = *s++;
  }
  return n;
}
```

```
16
17 struct command *
18 parse_cmd (const char *buf, int len)
19 {
20   struct command *cmd = NULL;
21   char prev = ' ';
22   int n, beg = 0;
23   if ((n = count_args(buf)) == 0) return NULL;
24   cmd = malloc(sizeof(*cmd));
25   cmd->argc = 0;
26   cmd->argv = malloc((n + 1) * sizeof(*cmd->argv));
27   for (int i = 0; i < len; i++) {
28     if (!isspace(buf[i]) && isspace(prev)) { beg = i; }
29     else if (isspace(buf[i]) && !isspace(prev)) {
30       cmd->argv[cmd->argc++] = strndup(buf + beg, i - beg);
31     }
32     prev = buf[i];
33   }
34   cmd->argv[cmd->argc] = NULL;
35   return cmd;
36 }
```

7. → Copiez ce code dans votre fichier, et écrivez une fonction `free_cmd` qui permet de libérer la mémoire d'un pointeur `struct command` pris en argument.
8. Il s'agit maintenant d'utiliser les fonctions `parse_cmd` et `free_cmd` dans notre boucle principale, de sorte à pouvoir par la suite facilement gérer les commandes avec des arguments.
   → Mettez à jour la boucle principale (en particulier le code de la question 5) pour utiliser ces nouvelles fonctions et la structure `command`.
9. → Enfin, ajoutez une commande intégrée `cd`, qui fait ce qui est attendu d'elle.
   Pensez à tester votre code et à corriger les erreurs.

Exercice 2.

Commandes simples.

1. Le but de cet exercice est de permettre à notre shell d'appeler des commandes existantes.
   → Étant donnée notre `struct command`, quelle fonction `exec*` va-t-on utiliser ?
2. → Écrivez une fonction `exec_cmd` qui prend en argument un pointeur vers une `struct command`, lance un nouveau processus exécutant cette commande et attend dans le processus du shell que celle-ci ait terminée. Cette fonction renvoie le code de retour de la commande exécutée.
3. → Dans la boucle principale, faite appel à `exec_cmd` si la commande n'est pas une des commandes intégrées.
4. Pour l'instant lorsqu'on envoie un signal d'interruption pendant l'exécution d'une commande, notre shell termine également.
   → Désactivez le gestionnaire par défaut du signal d'interruption pour votre shell, sans oublier de le réactiver pour les processus enfants. Pensez à tester par exemple avec la commande `cat` que votre code fonctionne correctement.

Exercice 3.

Commandes composées.

1. Dans cet exercice, on va ajouter le support des commandes composées de deux appels qui s'enchaînent. La syntaxe pour ces commandes sera "`commande1 | commande2`". Pour se simplifier la tâche on se limitera à deux commandes. Il s'agira alors d'exécuter les deux commandes en parallèle en connectant la sortie standard de la première dans l'entrée standard de la seconde.
   La première chose à faire est d'être capable de repérer si il y a un "`|`" sur la ligne de commande lue, et dans ce cas de pouvoir parser séparément les deux commandes.
   La fonction `strchr` de la bibliothèque C standard prend une chaîne de caractères et un caractères en argument. Elle renvoie un pointeur vers la position du caractère si elle le trouve, et `NULL` sinon.
   → Dans la boucle principale de notre shell, une fois la ligne de commande lue , récupérez dans une variable `pipe` la valeur de retour de `strchr` appelée avec la ligne de commande et le caractère "`|`".
2. Toujours pour des questions de simplicité, on va traiter séparément le cas des commandes simples et celui des commandes composées.
   Si la commande est simple, on garde le traitement actuel. Si elle est composée, on doit alors découper la ligne de commande en deux, et on supposera que cela ne contient pas de commandes intégrées.
   Le cas des commande simple étant déjà traité, on s'occupe maintenant du cas des commandes composées.
   → Si le buffer qui contient la ligne de commande s'appelle `buf`, que sa longueur est `len`, et que le caractère "`|`" dans `buf` est pointé par `pipe`, comment appeler notre fonction `parse_cmd` pour parser la première commande ? Et la seconde ?

3. Il ne nous reste plus qu'à écrire une fonction `exec_piped_cmds` qui prend deux pointeurs vers des `struct command` en argument et s'occupe de leur exécution enchaînée.

   Dans cette fonction il va falloir :
   — créer un tube (`pipe`),
   — créer un nouveau processus enfant dans lequel on dupliquera le côté écriture du tube sur la sortie standard et on fermera le côté lecture avant d'exécuter la première commande (`dup2`, `close`),
   — créer un nouveau processus enfant dans lequel on dupliquera le côté lecture du tube sur l'entrée standard et on ferma le côté écriture avant d'exécuter la seconde commande,
   — fermer dans le parent les deux côtés du tube,
   — attendre que les deux enfants aient terminé (`wait`),
   — renvoyer le code de retour de la deuxième commande.
   → Écrivez cette fonction en oubliant pas de gérer les erreurs.

4. → Appelez la fonction `exec_piped_cmds` dans la boucle principale, sans oublier de libérer la mémoire des deux `struct command`.

5. → Donnez quelques exemples de commandes permettant de tester votre code (et faites le !).