Université Joseph Fourier, Grenoble
Licence d'Informatique L3

Introduction aux Systèmes et Réseaux

TD n°4 : Organisation d'un *shell* (interprète de commandes)

L'objectif de ce TD [1] est d'examiner l'organisation d'un *shell* très rudimentaire, pour préparer le projet de mini-*shell* (TP-4 et Apnée-1).

1 Organisation d'un *shell*

Le programme principal d'un *shell* ressemble à celui de tout interprète :

```
while (TRUE) {
  lire commande;
  analyser commande;
  interpréter commande;
}
```

Dans le cas présent, le programme principal `myshell.c` est donné ci-après.

```
1   #include "myshell.h"
2   #define TRUE 1
3   // fonctions externes :
4   void eval(char*cmdline);
5   int parseline(char *buf, char **argv);
6   int builtin_command(char **argv);
7
8   int main() {
9       // la constante MAXLINE est définie dans csapp.h
10      char cmdline[MAXLINE];                  // ligne de commande
11
12      while (TRUE) {                          // boucle d'interprétation
13          printf("<my_shell> ");              // message d'invite
14          Fgets(cmdline, MAXLINE, stdin);     // lire commande
15          if (feof(stdin))                    // fin (control-D)
16              exit(0);
17          eval(cmdline);                      // interpréter commande
18      }
19  }
```

Le fichier d'inclusion "`myshell.h`" est :

```
#include "csapp.h"
#define MAXARGS 128
```

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

Les opérations `analyser commande` et `interpréter commande` sont réalisées respectivement par les programmes `parseline.c` et `eval.c` donnés ci-après. Le code source de ces programmes est dans le placard du TP4. Voici `eval.c` :

```
1   // eval : interprète une ligne de commande passée en paramètre
2   #include "myshell.h"
3
4   void eval(char *cmdline) {
5     char *argv[MAXARGS]; // argv pour execve()
6     char buf[MAXLINE];   // contient ligne de commande modifiée
7     int bg;              // arrière-plan ou premier plan ?
8     pid_t pid;           // process id
9
10    strcpy(buf, cmdline);
11    bg = parseline(buf, argv); // init argv et indique si tache d'arriere-plan
12    if (argv[0] == NULL)
13      return;                         // ignorer lignes vides
14
15    if (!builtin_command(argv)) {    // commande intégrée ?
16                                     // si oui, exécutée directement
17      if ((pid = Fork()) == 0) {     // si non, exécutée par un fils
18        if (execve(argv[0], argv, environ) < 0) {
19          printf("%s: Command not found.\n", argv[0]);
20          exit(0);
21        }
22      }
23
24      if (!bg) {   // le père attend fin du travail de premier plan
25        int status;
26        if (waitpid(pid, &status, 0) < 0)
27          unix_error("waitfg: waitpid error");
28      }
29      else         // travail d'arrière-plan, on imprime le pid
30        printf("%d %s", pid, cmdline);
31    }
32    return;
33  }
34
35  // si le premier paramètre est une commande intégrée,
36  // l'exécuter et renvoyer "vrai"
37  int builtin_command(char **argv)
38  {
39      if (!strcmp(argv[0], "quit"))   // commande "quitter"
40          exit(0);
41      if (!strcmp(argv[0], "&"))      // ignorer & tout seul
42          return 1;
43      return 0;                       // ce n'est pas une commande intégrée
44  }
```

On notera que l'on distingue les commandes dites "intégrées" (*built-in*), c'est-à-dire faisant partie intégrante du *shell* et interprétées par le processus qui exécute le *shell*, des

autres commandes, dont le code est contenu dans un fichier et qui sont exécutées par un processus séparé.

Le programme `parseline.c` qui analyse une ligne de commande est donné ci-après :

```
1   // parseline - analyse ligne de commande, construit tableau argv[]
2   #include "myshell.h"
3
4   int parseline(char *buf, char **argv) {
5     char *delim;         // pointe vers premier délimiteur espace
6     int argc;            // nombre d'arguments
7     int bg;              // travail d'arrière-plan ?
8     buf[strlen(buf)-1] = ' ';  // remplacer '\n' final par espace
9     while (*buf && (*buf == ' ')) // ignorer espaces au début
10      buf++;
11    argc = 0;
12    while ((delim = strchr(buf, ' '))) {   // construire liste args
13      argv[argc++] = buf;
14      *delim = '\0';
15      buf = delim + 1;
16      while (*buf && (*buf == ' '))        // ignorer espaces
17        buf++;
18    }
19    argv[argc] = NULL;                     // termine liste d'args
20    if (argc == 0)                         // ignorer ligne vide
21      return 1;
22    if ((bg = (*argv[argc-1] == '&')) != 0) // travail arrière-plan ?
23      argv[--argc] = NULL;
24    return bg;              // 1 si travail d'arrière-plan, 0 sinon
25  }
```

2 Préparation du projet

2.1 Question préliminaire

On demande d'abord de lire attentivement ces programmes et de comprendre leur fonctionnement. Quel défaut trouvez-vous à ce *shell* (autre que les limitations dues à sa simplicité) ? Comment le corriger ?

2.2 Introduction au projet

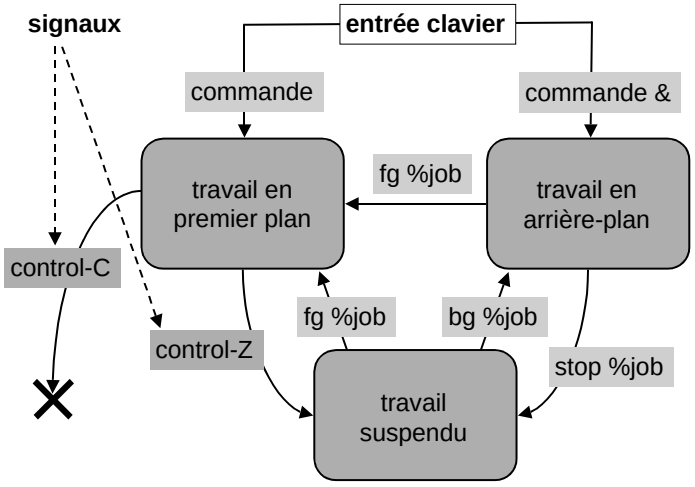
Le but du mini-projet est de construire un *shell* plus complet, partant des programmes ci-dessus. Ce *shell* exécutera les fonctions suivantes (analogues à celles des *shells* Unix courants) :

– interpréter des commandes, comme le *shell* initial. Les commandes intégrées sont exécutées par le processus shell lui-même, les autres (externes) par un fils. Si une commande externe se termine par `&`, elle est exécutée en arrière-plan, sinon en premier plan.

– Les commandes exécutées en arrière-plan s'appellent des *jobs* et sont désignées par un numéro de *job*. Un *job* peut être désigné par le `PID` du (groupe de) processus qui l'exécute (exemple 14567) ou par son numéro de *job* précédé de `%` (exemple `%3`).
– La frappe de *control-C* et *control-Z* doivent respectivement envoyer un signal `SIGINT` et un signal `SIGTSTP` au (groupe de) processus de premier plan. Les traitants par défaut de `SIGINT` et `SIGTSTP` consistent respectivement à terminer et à suspendre le processus destinataire.
– La commande intégrée `jobs` doit lister tous les *jobs* avec leur état, le nom de la commande qu'ils exécutent et le `PID` du processus qui les exécute. Exemple :

```
<myshell> jobs
[1] 3456  Stopped   commande1 hello 50
[2] 3458  Running   autrecommande
<myshell>
```

– Les commandes intégrées `fg` et `bg` s'appliquent à un *job*, désigné soit par son numéro de *job* soit par son numéro de processus. 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. La figure ci-après (cf. cours n°2) indique les états des travaux et les transitions entre ces états.



– Le *shell* doit ramasser tous ses fils zombis. Si un *job* se termine parce qu'il reçoit un signal ayant pour effet de le tuer, le *shell* doit imprimer un message indiquant le numéro du *job* et le numéro du signal.

2.3 Points à examiner

On examinera le principe de la réalisation des fonctions décrites ci-dessus. On distingue 3 aspects principaux :
– Structures de données et fonctions auxiliaires nécessaires pour la gestion des travaux.
– Programmation des traitants des principaux signaux utilisés.
– Interprétation des commandes intégrées.

Pour chacun de ces aspects, on pourra identifier les problèmes et proposer des ébauches de solutions, sans aller dans tous les détails, qui seront traités lors de la réalisation.