

Licence S&T Informatique, Univ. Bordeaux 1
Programmation Système — Année 2008-2009 — Projet

1 Modalités

Le projet est à réaliser par équipes de 3 étudiants, ou 4 si l'une des deux extensions optionnelles est choisie. Les sources du projet, ainsi qu'un court rapport en pdf (maximum 5 pages) détaillant les choix effectués et les résultats obtenus sont à transmettre par mail à l'ensemble des enseignants au plus tard le **12/12/2008**. Chaque étudiant d'un groupe présentera en soutenance la partie du travail qu'il a réalisée (on pourra demander de commenter le code). Les soutenances auront lieu **début janvier (semaine du 5/1/09)**.

2 Documents fournis

Le travail demandé est d'écrire un petit *shell*. Pour analyser la ligne de commande, un analyseur syntaxique écrit en `lex/yacc` est fourni. Il est rudimentaire, et il est conseillé de le modifier en cas de besoins spécifiques. Quelques détails sont donnés en section 5.

Le programme devra être écrit en langage C et pouvoir être compilé par `make`. Un fichier `Makefile` est fourni. Les options `-std=c99` et `-Wall` de `gcc` devront être conservées.

3 Travail demandé

Écrire un *shell* permettant d'interpréter des expressions et lançant des processus pour les exécuter. La grammaire suivante décrit la forme minimale des expressions qui doivent être traitées (ε désigne le mot vide). Il est possible d'ajouter d'autres constructions.

```
expression →  $\varepsilon$ 
|
| commande
| expression ; expression
| expression || expression
| expression && expression
| ( expression )
| expression | expression
| expression > fichier
| expression 2> fichier
| expression &> fichier
| expression < fichier
| expression >> fichier
```

La signification des différents symboles est la même qu'avec un shell usuel, comme `zsh`. Une commande pourra être soit interne, soit externe, et peut évidemment comporter des arguments.

Les commandes internes suivantes, empruntées à `zsh/bash` ou `csh` devront être interprétées avec leur signification habituelle : `builtins`, `alias`, `unalias`, `cd`, `dirs`, `popd`, `pushd`, `echo`, `exec`, `exit`, `history`, `kill`, `printenv`, `pwd`, `source`, `umask`, `times`.

4 Extensions

On pourra écrire au choix l'une des extensions suivantes¹ (par groupe de 4 étudiants).

1. Ajout du *job control*. Il s'agit d'ajouter les fonctionnalités pour
 - lancer des processus en arrière plan, par *expression &*.
 - stopper par `Ctrl-Z` le (groupe des) processus en premier plan,
 - envoyer à ce groupe des signaux générés au terminal (comme `SIGINT` généré par `Ctrl-C`).
 - ajouter les commandes internes `bg`, `fg`, `jobs`, `disown`, `wait`. Le job `i` sera désigné par `%i`.
2. Manipulation de l'environnement, en ajoutant la commande interne `export`, et complétion :
 - des variables d'environnement lorsqu'elles sont précédées du caractère `$` (on ne demande cependant pas d'écrire un mécanisme général de complétion).
 - des noms de fichiers. Le shell devra interpréter les caractères spéciaux courants : `*`, `?`, `\...`

5 Annexe : l'analyseur syntaxique fourni

L'analyseur syntaxique fourni permet de transformer une expression entrée par l'utilisateur en arbre représentant cette expression. En cas d'analyse correcte, la variable globale `ExpressionAnalysee` pointe sur un arbre représentant l'expression. Le type `Expression` de cet arbre est décrit dans le fichier `Shell.h`. Il contient 4 champs. Si `e` est du type `Expression` :

- `e.type` est un type d'expression, contenant une valeur définie dans `Shell.h`. Cette valeur peut être par exemple (voir le fichier `Shell.c` pour une liste complète) :
 - `SIMPLE`, représentant une commande simple et ses arguments.
 - `SEQUENCE`, représentant une séquence (`;`).
 - `PIPE`, représentant un *pipe* (`|`).
 - `REDIRECTION_I`, représentant une redirection de l'entrée (`<`).
- `e.gauche` et `e.droite`, de type `Expression *`, représentent une sous-expression gauche et une sous-expression droite. Ces deux champs ne sont pas utilisés pour les types `VIDE` et `SIMPLE`. Pour les expressions réclamant deux sous-expressions (comme `PIPE`) ces deux champs sont utilisés simultanément. Pour les types qui ne réclament qu'une sous-expression, comme `REDIRECTION_IN`, seule l'expression gauche est utilisée.
- `e.arguments`, de type `char **`, a deux interprétations :
 - pour une commande simple, `e.arguments` est un tableau à *la argv* terminé par `NULL`. Ainsi, `(e.arguments)[0]` est le nom de la commande, `(e.arguments)[1]` est le 1^{er} argument, etc. Ce tableau peut donc être passé directement à une fonction `execv[pe]()`.
 - si la commande est une redirection, `(e.arguments)[0]` est le nom du fichier vers lequel on redirige.

Par ailleurs, l'analyseur ne fait aucune désallocation tel qu'il est écrit, et la construction des tableaux `arguments` se fait via une zone statique (`NB_ARGS` arguments au plus) et la taille des identificateurs est limitée à `TAILLE_ID`. Ces deux valeurs valent 50 et 500, par défaut. Enfin, si le *shell* reçoit une fin de fichier (`^D` par défaut), la fonction `EndOfFile()` est appelée. Dans la version fournie, elle provoque la sortie du programme.

1. D'autres extensions (comme la duplication de descripteurs, les fonctions shell) peuvent être ajoutées si le temps/l'envie le permettent, mais pour tout groupe de 4, on demande d'écrire l'une des deux extensions proposées.