Projet de Compléments en Programmation Orientée Objet n° 2 : Petite invite de commandes (minishell)

I) Invite de commandes

Le but de ce projet est d'écrire un programme qui simule une invite de commande (shell) et permet d'exécuter quelques commandes de base.

- Le programme doit afficher des messages d'erreurs en français, comprenant le cas échéant des lettres accentuées.
- Il doit être fourni un fichier expliquant ce que vous avez fait, soit dans un fichier texte brut (README.txt) soit dans un fichier au format pdf.
- Une attention particulière doit être accordée au traitement des exceptions : le fait qu'un fichier n'existe pas ou qu'un argument soit mal formaté ne doit pas faire planter le programme.

II) Remarques générales

- Dans la suite du sujet, nous utilisons les chevrons (`<` et `>`) pour délimiter un argument obligatoire d'une commande et les crochets (`[` et `]`) pour délimiter un argument ou portion de la commande qui est optionnel. Ces caractères ne servent qu'à décrire la syntaxe d'une commande et ne doivent en aucun cas être écrits par l'utilisateur (on parle de métasyntaxe).
- Vous pouvez comparer vos commandes avec celles du shell classique. Il faut cependant faire attention à deux choses : bash interprète certains caractères (typiquement `*`) *avant* de donner les arguments à la commande et il faut donc rajouter des côtes (`'`) autours des expressions régulières utilisées dans bash. Les expressions régulières ne sont pas complètement standard et il se peut donc que les résultats de bash pour grep ou find soient parfois différents de ceux donnés par vôtre implémentation (ce n'est pas grave !). On ne vous demande pas de ré-implémenter les expressions régulières.

III) Commandes obligatoires

a) Pour s'échauffer

Implémentez les commandes suivantes :

- `ls`
  Affiche la liste des dossiers et fichiers contenus dans le répertoire courant.
- `ps`
  Affiche la liste des commandes en cours d'exécution et leur `pid`.
- `pwd`
  Affiche le nom complet du répertoire courant.
- `cd <sous-répertoire>`
  Change le répertoire courant, le sous-répertoire doit-exister. L'argument spécial `..` permet de remonter au sur-répertoire.

- `date [<format>=+%Y-%m-%d]`
  Cette méthode prend soit zéro, soit un argument qui commence [1] par un '+' et qui donne le format de la date à afficher. Si aucun argument n'est donné utiliser le format `+%Y-%m-%d`. La commande affiche la chaîne décrite par le format en remplaçant certaines sous-chaînes comme indiqué ci-dessous.
  - `%d` : jour du mois (01-31)
  - `%H` : heure /24 (0-23)
  - `%m` : mois (01-12)
  - `%M` : minute (00-59)
  - `%Y` : année (ex : 2015)

  (Voir la classe `SimpleDateFormat` dans la documentation de java pour obtenir ces différentes valeurs.)
- `find <chemin> -name <expr. reg.>`
  `find <chemin> -iname <expr. reg.>`
  Écrit sur la sortie standard le nom de tous les fichiers qui sont dans le répertoire `<chemin>` (ou un de ses sous-répertoires) et dont le nom *top-level* vérifie l'expression régulière donnée en argument. La variante utilisant `-iname` ne tient pas compte de la casse.

b) Multi-tâche de base

À partir de maintenant, on manipule plusieurs threads (pour un même terminal) : chaque commande se lance dans un nouveau thread (bifurqué dès qu'on valide) [2]. À chaque thread est attribué un `pid` (process id) unique (c'est juste un entier). Ce `pid` est affiché dans le terminal dès que le thread est instancié.

- Dans le but de tester, en simulant l'exécution d'une commande longue, ajoutez la commande `compteJusqua <entier> [<format>=%d\n]`
  qui pour argument(s) : un nombre entier (obligatoire) et un format (optionnel). Si ce dernier n'est pas précisé, on prend par défaut le format `%d\n`.
  Cette commande affiche le format une fois par seconde en remplaçant l'occurrence éventuelle du premier %d par la seconde courante.
  Ainsi, utilisée avec un seul argument, cette commande affiche les `<entier>` plus petits nombres entiers dans l'ordre et à raison d'un par seconde.
- Implémenter la commande `kill <pid>` qui arrête prématurément la commande dont le pid est donné en argument.

IV) Améliorations

a) Commandes additionnelles

- `grep <expr. reg.> [ <fich. 1> [ <fich.2> [ ... ] ] ]`
  Prend un argument obligatoire et n'importe quel nombre de chemin vers des fichiers. Cherche dans tous les fichiers donnés en arguments quels sont les lignes qui contiennent l'expression régulière `<expr. reg>`. Si aucun argument n'est donné, la commande lit les lignes depuis l'entrée standard jusqu'à la lecture d'un caractère "fin de fichier" (EOF) (que l'on écrit sous linux en tapant `CTRL+D`).

1. Ce '+' est parfaitement inutile dans notre cas, il est là pour s'approcher de la syntaxe de la commande `date` du shell POSIX.
2. Comme si on avait suivi la commande de "&" dans le shell Unix.

- `sed <format> [<fichier>]`
  La commande `sed` est principalement utilisée pour faire des remplacements de chaîne dans un fichier, c'est la seule utilisation qu'on demande d'implémenter ; si aucun fichier n'est donné (c'est-à-dire si la commande n'a qu'un argument), alors l'entrée standard doit-être lue à la place ; La chaîne de format est de la forme
  `s<char><regexp1><meme char><chaine1><meme char><chaine2>`
  Le caractère de séparation est le deuxième caractère de `<format>` et ne peut pas apparaître dans les autres chaines.
  La commande affiche donc le fichier (ou l'entrée standard) en remplaçant, dans chaque ligne, la première occurrence de `<regexp1>` par `<chaine1>` ; si `<chaine2>` est égale à `"g"` toutes les occurrences sont remplacées à la place, sinon `<chaine2>` est ignorée. Par exemple, les commandes
  `sed s/Hello/World/g` et `sed s:Hello:World:g`
  font la même chose, à savoir écrire sur la sortie standard chaque ligne écrite sur l'entrée standard en remplaçant chaque occurrence de `Hello` par `World`.

b) Multi-tâche avancé

On retourne au comportement par défaut du début du projet : une commande s'exécute (par défaut), en avant-plan dans le thread courant.

- Faire en sorte d'exécuter la commande dans un thread différent seulement si elle est suivie du caractère `&`.
- Ajouter le support du `;` (point-virgule) : deux commandes ou plus sur une même ligne, séparées par un point-virgule, s'exécutent l'une après l'autre. Attention, & est prioritaire par rapport à `;` : `a; b &` exécute `a` (dans le thread courant), puis `b` en arrière plan (nouveau thread).
- Ajouter le support des parenthèses, de telle sorte que `(a; b; c) &` exécute en arrière plan toute la séquence `a`, puis `b` puis `c`.
- Implémenter la commande `sleep <n>` qui met en pause la tâche qui appelle cette commande pendant n secondes.
- Implémenter la commande `wait` qui bloque la tâche courante en attendant que tous les autres travaux en cours d'exécution dans ce terminal soient terminés.

c) Redirections de flux

- Implémenter le "pipe" ('|'), qui fournit la sortie standard d'une commande comme entrée standard de l'autre. Par exemple la commande
  `find . -name *txt | grep "Bonjour"`
  affiche une ligne d'un fichier si cette ligne contient le mot "bonjour" et que le nom du fichier termine par `"txt"`.
- Implémenter la redirection de sortie (`">"` et `">>"`) qui écrit la sortie standard d'une commande dans un fichier. La commande
  `find . -name *txt >> find.log`
  écrit les noms de tous les fichiers dont le nom termine par `"txt"` dans le fichier `find.log`. Le fichier est préalablement effacé si l'on utilise `">"` mais pas si on utilise `">>"`.