TP Systèmes 2 : Mini-shell

Ensimag 2A

Édition 2023-2024

Le but de ce TP est de vous familiariser avec la gestion des processus en vous faisant écrire un petit interpréteur de commandes, sous ensemble de ce que vous utilisez tous les jours (bash, tcsh, zsh, ou même le PowerShell de Windows 7/8/10).

La démarche du sujet est progressive : le *shell* est enrichi au fur et à mesure de nouvelles fonctionnalités. Le socle est commun. Différentes variantes sur les fonctionnalités sont définies (cf. section 6).

1 Compilation, test unitaires et rendu

Le squelette fourni inclut un script `cmake` pour construire automatiquement les Makefile utiles.

Vous devez préalablement modifier le début du fichier `CMakeLists.txt` votre numéro de variante. En remplaçant le -1, par votre numéro de variante à la ligne

```
set(VARIANTE_SUJET -1)
```

Vous devez préalablement également créer et remplir un fichier `AUTHORS`, à la racine du projet, avec vos prénoms, noms et logins Typiquement pour l'équipe Alice Liddell et Bob Morane, cela donne un fichier AUTHORS

```
Alice Liddell liddella
Bob Morane moraneb
```

1.1 Attendus

Acceptable : Faire l'implémentation correcte du shell de base (redirection, pipe, tâche de fond).

Bien : Faire l'implémentation correcte du shell et de votre variante.

Très bien : Faire l'implémentation complète avec l'interpréteur Scheme, en ayant refactoriser votre code pour ne pas avoir de duplication.

Excellent : Faire l'implémentation de toutes les variantes

1.2 Compilation

La création des Makefile s'effectue en utilisant `cmake` dans un répertoire dans lequel seront aussi créés les fichiers générés. Le répertoire "build" du squelette sert à cet usage. Tout ce qui apparaît dans "build" pourra donc être facilement effacé ou ignoré.

La première compilation s'effectue alors en tapant :

```
$ cd ensimag-shell
$ cd build
$ cmake ..
$ make
$ make test
```

et les suivantes, dans le répertoire `build/`, avec

```
make
make test
make check
```

1.3 Tests unitaires

Une batterie de test vous est fournie pour vous aider à construire plus rapidement un code correct. `make test` devrait donc trouver la plupart des bugs de votre programme au fur est à mesure que vous l'écrivez. Vous pouvez lancer directement l'exécutable `../tests/allShellTests.rb` avec `make check` pour avoir plus de détails. Vous pouvez aussi lancer individuellement les tests qui vous intéressent (cf. `../tests/allShellTests.rb --help`).

1.4 Rendu des sources

L'archive des sources que vous devez rendre dans `Teide` est généré par le makefile créé par cmake :

```
$ cd ensimag-shell
$ cd build
$ make package_source
```

Il produit dans le répertoire `build`, un fichier ayant pour nom (à vos login près) `Ensishell-1.0.login1-login2-login3-Source.tar.gz`.

C'est ce fichier tar qu'il faut rendre.

1.4.1 Mon fichier TAR est trop gros pour Teide !

Si vous avez ajouté des fichiers binaires dans votre historique `.git`, ou bien que vous utilisez VS Code, ou bien un éditeur de Jet Brain, vous aurez peut-être des répertoires cachés `.git`, `.vscode` ou `.idea` qui font grossir votre TAR.

Vous pouvez éditer la liste des fichiers exclus dans le `CMakeLists.txt` avant de créer le fichier TAR. Ou bien, à postériori, vous pouvez éditer le TAR, soit en ligne de commande avec gunzip, `tar --delete` et gzip, soit avec un gestionnaire de fichiers comme nautilus, thunar ou emacs, ou un gestionnaire d'archive comme ark, emacs ou file-roller.

2 Lancement et enchaînement de commandes

Pour commencer, vous devez simplement lire des commandes sur l'entrée standard et les exécuter. Pour cela vous devez utiliser les appels système `fork(...)` et `exec(...)` (`man 2 fork` , `man 3 execvp`).

Exemple d'utilisation :

```
$ /bin/ls /usr
X11R6  etc    include  kerberos  lib64    local  share  tmp
bin    games  java     lib       libexec  sbin   src
$ pwd
/perms/denneuli
```

Question 1 (Lancement d'une commande) *Écrire le programme qui lit des commandes sur l'entrée standard et les exécute. Pour vous aider, vous disposez d'une fonction de lecture de l'entrée standard (fichier* readcmd.c*, exemple d'utilisation dans* ensishell.c*).* readcmd.c *utilise la bibliothèque libre* readline *pour vous permettre l'édition des lignes, la complétion avec le* tab *et la sauvegarde de l'historique.*

2.1 Lancement en tâche de fond

Lancer une commande dont le temps d'exécution est assez long (`sleep 20` par exemple). Vous constaterez que le prompt apparaît immédiatement, sans attendre la fin de l'exécution. En effet, par défaut, le processus père n'attend pas son fils, celui qu'il a créé. Pour cela il faut utiliser la routine `wait` ou `waitpid` [1].

Question 2 (Attente de la terminaison) *Modifier votre programme pour qu'il attende la fin d'une commande lancée avant de passer à la suivante.*

Question 3 (Tâche de fond) *Ajouter ensuite la possibilité de lancer une commande en tâche de fond si besoin. Pour cela, le caractère & en entrée positionne le champ* `bg` *dans la* `struct cmdline` *.*

Exemple :

```
$ du -s /tmp &
$ ls
....
$
278200  /tmp
```

Le parseur fournit ne reconnaît le caractère & que s'il est seul. Il faut un espace entre le & et l'argument le précédant.

1. Des détails sur l'appel `waitpid` en tapant `man 2 waitpid`

2.2 Liste des processus

Question 4 (Lister les processus en tâches de fond) *Ajouter à votre shell une commande interne* `jobs` *qui donne la liste des processus lancés en tâche de fond de votre shell avec leur pid et la commande lancée. Comment pouvez-vous savoir s'ils se sont terminés ? (* `man waitpid` *)*

3 Le pipe

Question 5 (pipe) *Ajouter la possibilité de connecter l'entrée d'un processus avec la sortie d'un autre comme dans la commande* `ls / | grep u` *.*

La question ne concerne que le cas avec 2 commandes seulement (1 seul pipe). Les appels systèmes utiles sont `pipe(...)`, `dup(...)`, `dup2(...)`, `close(...)`.

4 Redirection dans les fichiers

Question 6 (Redirection) *Ajouter la possibilité de connecter l'entrée ou la sortie d'un processus avec des fichiers comme dans la commande* `cat < toto > tata` *.*

Les redirections et le pipe peuvent être appelés ensemble. Les appels systèmes utiles sont `open(...)`, `dup(...)`, `dup2(...)`, `close(...)`.

4.1 Les fichiers déjà existants

Les fichiers déjà existant ne sont pas tronqués automatiquement par une ouverture en écriture. Par défaut, les écritures se superposent à ce qui est déjà écrit. Lors de la redirection en sortie vers un fichier déjà existant, il faudra, juste après l'avoir ouvert, réduire sa taille à 0 pour effacer son contenu précédent avec la fonction `ftruncate`.

5 Appel de l'interpréteur Scheme

Il est très facile d'embarquer un langage de programmation dans votre interpréteur (Perl, Python, Ruby, Lua, etc.) et de l'étendre. Dans le squelette, c'est quelques lignes de code. Un point dur pour notre shell est de différencier une commande shell des instructions du langage.

Le squelette fourni inclut un interpréteur `Scheme` (`https://fr.wikipedia.org/wiki/Scheme`), nommé `guile`. Scheme est un langage fonctionnel, version épurée du langage LISP (utilisé dans emacs). Toutes les expressions du langage commençant par une parenthèse ń `(` ż, le squelette différencie une commande shell d'une expression en Sheme en testant la première lettre d'une ligne.

Les instructions suivantes fonctionnent directement avec le squelette.

```
1 (display "Hello world !\n")
2 (display (+ 2 2))(newline)
```

L'interpréteur Scheme est étendue par la fonction `int question6_executer(char *)` que vous devez implanter. Cette fonction doit exécuter la commande shell passée dans la chaîne de caractère en argument.

Le code suivant doit créer un répertoire nommé `Toto`.

```
(executer "mkdir Toto")
```

Il suffit de refaire exactement ce que vous faites dans les questions précédentes. Si votre code est correctement découpé en fonction, le code à ajouter est trivial. Sinon, vous devriez changer sa structure.

Les appels systèmes utiles sont `fork(...)`, `execvp(...)`, `waitpid(...)`, `open(...)` mais ils ne devraient pas être appelés directement. La fonction `parsecmd(...)` sera elle aussi utile.

Vous devriez pouvoir exécuter le petit programme suivant qui crée des répertoires dont les noms sont les nombres de fibonnacci 10, 20 et 30.

```
(define (fibo n) (if (<= n 2) 1 (+ (fibo (- n 1)) (fibo (- n 2))))) (for-each
↪  (lambda (i) (executer (string-append "mkdir " (number->string (fibo
↪  i))))) '(10 20 30))
```

Dans notre interpréteur simple, le code Scheme doit tenir sur une seule ligne. Mais, il est possible de charger et d'évaluer un fichier Scheme contenant votre code.

```
(load "repertoires_fibo.scm")
```

6 Les variantes

La suite du sujet est fonction de votre choix de variante. Choisissez-là et pensez bien à insérer son numéro (cf section 1) pour faciliter la vie de votre correcteur.

ID	Variantes
0	Jokers et environnement (sec. 6.1) ; Limitation du temps de calcul (sec. 6.6)
1	Jokers étendus (tilde, brace) (sec. 6.2) ; Pipes multiples (sec. 6.5)
2	Terminaison asynchrone (sec. 6.4) ; Limitations du temps de calcul (sec. 6.6)
3	Temps de calcul (sec. 6.3) ; Pipes multiples (sec. 6.5)
4	Jokers et environnement (sec. 6.1) ; Pipes multiples (sec. 6.5)
5	Jokers étendus (tilde, brace) (sec. 6.2) ; Limitation du temps de calcul (sec. 6.6)
6	Terminaison asynchrone (sec. 6.4) ; Pipes multiples (sec. 6.5)
7	Temps de calcul (sec. 6.3) ; Limitation du temps de calcul (sec. 6.6))
8	Jokers et environnement (sec. 6.1) ; Terminaison asynchrone (sec. 6.4) ;
9	Jokers étendus (tilde, brace) (sec. 6.2) ; Temps de calcul (sec. 6.3) ;
10	Jokers et environnement (sec. 6.1) ; Temps de calcul (sec. 6.3) ;
11	Jokers étendus (tilde, brace) (sec. 6.2) ; Terminaison asynchrone (sec. 6.4) ;

TABLE 1 – Les différentes variantes

6.1 Les jokers et les variables d'environnements

Les jokers (l'étoile, les crochets, le tilde, les variables d'environnements) sont remplacés dans la ligne de commande par le shell avant l'exécution.

Question 7 (Joker en wordexp) *Ajouter la possibilité d'utiliser les jokers et les variables d'environnements comme dans la commande* `ls ~/t* \$PWD/toto` *.*

Les appels systèmes et fonctions utiles sont `wordexp(...)` , `wordfree(...)` , `strnlen(...)` , `strlcpy(...)` , `strlcat(...)` , `malloc(...)` , `free(...)` .

6.2 Jokers étendus

Les jokers sont remplacés dans la ligne de commande par le shell avant l'exécution.

Question 8 (Joker en glob) *Ajouter la possibilité d'utiliser les jokers comme dans la commande*
`ls ~/t{oto,iti} *.c`.

Les appels systèmes et fonctions utiles sont `glob(...)` , `globfree(...)` , `strnlen(...)` , `strlcpy(...)` , `strlcat(...)` , `malloc(...)` , `free(...)` .

6.3 Temps de calcul d'un processus

Lorsqu'un processus termine, si son père n'est pas en attente de sa terminaison, il lui envoie un signal `SIGCHLD`. La réception du signal peut déclencher un traitant qui réalise un certain nombre d'actions.

Question 9 (Signaux) *Le but est d'afficher un message indiquant le temps de calcul d'un processus lancé en tache de fond (avec un &) à sa terminaison. Le shell devra afficher immédiatement le message.*

Les appels systèmes et fonctions utiles sont `waitpid(...)` , `sigaction(...)` , `gettimeofday(...)` .

La réception d'un signal débloque les appels systèmes bloquants comme `waitpid(...)` ou `read(...)` (cf leurs manuels). Il faudra les reprendre le cas échéant.

6.4 Terminaison asynchrone

Le but est d'afficher la terminaison d'un processus s'exécutant en tâche de fond au moment où celui-ci se termine, sans attendre le prochain prompt de l'interpréteur demandant une commande.

Question 10 (Signaux) *Il faudra traiter le signal "SIGCHILD" reçu par le père (le shell) lorsque le fils termine. La fonction traitant associée au signal est exécutée de manière asynchrone dès que le signal arrive au processus.*

Les appels systèmes utiles sont `signal(...)` .

La réception d'un signal débloque les appels systèmes bloquants comme waitpid ou read (cf leurs manuels). Il faudra les reprendre le cas échéant.

6.5 Pipes multiples

Question 11 (Pipes multiples) *Le but est que les séquences de pipes multiples fonctionnent.*

```
$ ls -R / | egrep "^to" | egrep "\.jpg$" | gzip -c | gzip -cd | less
```

Les appels systèmes utiles sont `pipe(...)`, `dup(...)`, `dup2(...)`, `close(...)`.

6.6 Limitation du temps de calcul

Il est possible de fixer une limite maximum au temps de calcul que les processus lancés peuvent utiliser. À la fin d'un délai souple, le processus reçoit un signal SIGXCPU, qui par défaut le termine. S'il intercepte le signal, il le reçoit ensuite chaque seconde jusqu'à la fin du délai dur, où il est détruit par un signal SIGKILL.

Question 12 (ulimit) *Le but est d'implanter la commande interne* `ulimit X`*, où X est le nombre de secondes avant la limite souple. Mettez la limite dure à X+5 secondes plus tard.*

Attention, la limite ne doit s'appliquer qu'aux nouveaux processus lancés et pas au shell.
Les appels systèmes et fonctions utiles sont `setrlimit(...)`, `strcmp(...)`, `atoi(...)`,.