Projet : programmation d'un interpréteur de commandes

L3 Informatique - Système

Il est important de bien lire le sujet jusqu'au bout et de bien y réfléchir avant de se lancer dans la programmation du projet.

Sujet : `jsh`, un shell permettant la gestion des tâches (*job control*)

Le but du projet est de programmer un interpréteur de commandes (aka *shell*) interactif reprenant quelques fonctionnalités classiques des shells usuels, en particulier le *job control*, c'est-à-dire la gestion des tâches lancées depuis le shell. Outre la possibilité d'exécuter toutes les commandes externes, `jsh` devra proposer quelques commandes internes, permettre la redirection des flots standard ainsi que les combinaisons par tube, et adapter le prompt à la situation.

Plus précisément, `jsh` doit respecter les prescriptions suivantes :

Notion de job

Un terminal peut contrôler de nombreux processus simultanément, mais ne peut avoir, à un instant donné, qu'un seul groupe de processus à l'avant-plan. `jsh` permet le contrôle de l'exécution simultanée de plusieurs tâches, ainsi que leur bascule entre l'avant- et l'arrière-plan, en gardant en mémoire toutes les informations nécessaires à la surveillance, à l'interruption et à la reprise des processus concernés.

Pour chaque ligne de commande nécessitant la création d'un ou plusieurs processus supplémentaires, `jsh` crée un nouveau *groupe de processus*, qui constitue un *job*. Ce groupe reçoit alors un *numéro de job*, qui l'identifie de manière unique parmi les jobs en cours dépendant de l'instance courante de `jsh`.

`jsh` peut lancer de nouveaux jobs *à l'arrière-plan* ou *à l'avant-plan* du terminal, selon que la ligne de commande termine ou non par le symbole `&`. Il peut *basculer* un job de l'arrière-plan vers l'avant-plan, ou *relancer l'exécution d'un job suspendu*, vers l'avant- ou l'arrière-plan. Il surveille les jobs à l'arrière-plan et annonce leur terminaison. Il peut lister les jobs surveillés en précisant leur état :

- un job dont tous les processus sont terminés, et tel que tous les processus directement lancés par le shell ont correctement terminé (avec `exit`), est dit *achevé* (*done*);
- un job dont tous les processus sont terminés, et tel qu'au moins processus directement lancé par le shell a terminé suite à la réception d'un signal, est dit *tué* (*killed*);
- un job dont au moins un processus n'a pas terminé, mais dont tous les processus directement lancés par le shell ont terminé, est *détaché* (*detached*) et quitte la surveillance de `jsh`;
- un job dont tous les processus non terminés directement lancés par le shell sont suspendus est lui-même dit *suspendu* (*stopped*);
- les autres jobs sont dits *en cours d'exécution* (*running*).

Un job sort de la liste des jobs surveillés lorsque sa terminaison a été annoncée, ou qu'il est détaché.

Un job est désigné par son numéro, précédé du symbole `%` pour lever toute ambiguïté avec un pid.

Commandes externes

`jsh` peut exécuter toutes les commandes externes, avec ou sans arguments, en tenant compte de la variable d'environnement `PATH`.

Commandes internes

`jsh` possède (au moins) les commandes internes suivantes :

`pwd`

Affiche la référence physique absolue du répertoire de travail courant.

La valeur de retour est 0 en cas de succès, 1 en cas d'échec.

`cd [ref | -]`

Change de répertoire de travail courant en le répertoire `ref` (s'il s'agit d'une référence valide), le précédent répertoire de travail si le paramètre est `-`, ou `$HOME` en l'absence de paramètre.

La valeur de retour est 0 en cas de succès, 1 en cas d'échec.

`?`

Affiche la valeur de retour de la dernière commande exécutée.

`exit [val]`

Si un ou plusieurs jobs sont en cours d'exécution ou suspendus, affiche un message d'avertissement; dans ce cas, la valeur de retour est 1, et `jsh` affiche une nouvelle invite de commande.

Sinon, termine le processus `jsh` avec comme valeur de retour `val` (si un argument est fourni), ou par défaut la valeur de retour de la dernière commande exécutée.

Lorsqu'il atteint la fin de son entrée standard (ie si l'utilisateur saisit `ctrl-D` en mode interactif), `jsh` se comporte comme si la commande interne `exit` (sans argument) avait été saisie.

`jobs [-t] [%job]`

Sans option ni argument, affiche la liste des jobs en cours, en précisant pour chacun :

- le numéro de job, entre crochets;
- l'identifiant du groupe de processus;
- l'état du job (Running, Stopped, Detached, Killed ou Done);
- la ligne de commande qu'il exécute.

Par exemple :

```
[1] 590622  Done     sleep 1000
[2] 590643  Running  sleep 500
[3] 590664  Stopped  vim toto.c
[4] 591141  Running  ./a.out | wc -l > /tmp/tutu
```

Avec l'option `-t`, liste **l'arborescence** des processus de chaque job, en indiquant pour chacun son pid, son état et la commande qu'il exécute.

Par exemple :

```
[2] 590643  Done         sleep 500
[3] 590664  Stopped      vim toto.c
[4] 591141  Running      ./a.out
    | 591143  Running    ./a.out
    591142  Running      wc -l > /tmp/tutu
```

Si un numéro de job est passé en argument à `jobs`, la liste est restreinte au job concerné.

`bg %job`

Relance à l'arrière-plan l'exécution du job spécifié en argument.

`fg %job`

Relance à l'avant-plan l'exécution du job spécifié en argument.

`kill [-sig] %job` **ou** **`kill [-sig] pid`**

Envoie le signal `sig` (ou `SIGTERM` par défaut) à tous les processus du job de numéro `job`, ou au processus d'identifiant `pid`.

Par exemple, `kill -9 5312` envoie `SIGKILL` (9) au processus de pid 5312, tandis que `kill %2` envoie `SIGTERM` à tous les processus du job numéro 2.

Gestion de la ligne de commande

`jsh` fonctionne en mode interactif : il affiche une invite de commande (*prompt*), lit la ligne de commande saisie par l'utilisateur, la découpe en mots selon les (blocs d')espaces, interprète les éventuels caractères spéciaux (`&`, `<`, `>`, `|`, cf ci-dessous), puis exécute la ou les commandes correspondantes.

Avant chaque nouvel affichage de l'invite de commande, `jsh` effectue un tour des jobs en cours, et affiche le cas échéant, sur la sortie d'erreur, la liste des jobs ayant changé de statut (job créé, stoppé, terminé ou détaché). Exception : `jsh` n'affiche rien pour un job qui était à l'avant-plan et dont le nouveau statut est *achevé*, *tué* ou *détaché*.

`jsh` ne fournit aucun mécanisme d'échappement; les arguments de la ligne de commande ne peuvent donc pas contenir de caractères spéciaux (espaces en particulier).

Il est fortement recommandé d'utiliser la bibliothèque `readline` pour simplifier la lecture de la ligne de commande : cette bibliothèque offre de nombreuses possibilités d'édition pendant la saisie de la ligne de commande, de gestion de l'historique, de complétion automatique... Sans entrer dans les détails, un usage basique de la fonction `char *readline (const char *prompt)` fournit déjà un résultat très satisfaisant :

```
char * ligne = readline("mon joli prompt $ ");
/* alloue et remplit ligne avec une version "propre" de la ligne saisie,
 * ie nettoyée des éventuelles erreurs et corrections (et du '\n' final) */
add_history(ligne);
/* ajoute la ligne à l'historique, permettant sa réutilisation avec les
 * flèches du clavier */
```

L'affichage du prompt de `jsh` est réalisé **sur sa sortie erreur**. Avec la bibliothèque `readline`, ce comportement s'obtient en indiquant `rl_outstream = stderr;` avant le premier appel à la fonction `readline()`.

Formatage du prompt

Le prompt est limité à 30 caractères (apparents, ie. sans compter les indications de couleur), et est formé des éléments suivants :

- un premier code de bascule de couleur;
- entre crochets, le **nombre de jobs** actuellement surveillés;
- une deuxième bascule de couleur;
- la référence du **répertoire courant**, éventuellement tronquée (par la gauche) pour respecter la limite de 30 caractères; dans ce cas la référence commencera par trois points (`"..."`);
- la bascule `"\033[00m"` (retour à la normale);
- un dollar puis un espace (`"$ "`).

Exemples de codes de bascule de couleur : `"\033[32m"` (vert), `"\033[33m"` (jaune), `"\033[34m"` (bleu), `"\033[36m"` (cyan), `"\033[91m"` (rouge). Pour que l'affichage s'adapte correctement à la géométrie de la fenêtre, chaque bascule de couleur doit être entourée des caractères-balises `'\001'` et `'\002'` (qui indiquent que la portion de chaîne qu'ils délimitent est formée de caractères non imprimables et doit donc être ignorée dans le calcul de la longueur du texte à afficher).

Par exemple (sans coloration) :

```
[0]/home/titi$ cd pas/trop/long
[0]/home/titi/pas/trop/long$ cd mais/la/ca/depasse
[0]...ong/mais/la/ca/depasse$ pwd
/home/titi/pas/trop/long/mais/la/ca/depasse
[0]...ong/mais/la/ca/depasse$ sleep 1000 &
[1] 590522  Running  sleep 1000
[1]...ong/mais/la/ca/depasse$
```

Redirections

`jsh` gère les redirections suivantes :

- `cmd < fic` : redirection de l'entrée standard de la commande `cmd` sur le fichier (ordinaire, tube nommé...) `fic`
- `cmd > fic` : redirection de la sortie standard de la commande `cmd` sur le fichier (ordinaire, tube nommé...) `fic` **sans écrasement** (ie, échoue si `fic` existe déjà)
- `cmd >| fic` : redirection de la sortie standard de la commande `cmd` sur le fichier (ordinaire, tube nommé...) `fic` **avec écrasement** éventuel
- `cmd >> fic` : redirection de la sortie standard de la commande `cmd` sur le fichier (ordinaire, tube nommé...) `fic` **en concaténation**
- `cmd 2> fic` : redirection de la sortie erreur standard de la commande `cmd` sur le fichier (ordinaire, tube nommé...) `fic` **sans écrasement** (ie, échoue si `fic` existe déjà)
- `cmd 2>| fic` : redirection de la sortie erreur standard de la commande `cmd` sur le fichier (ordinaire, tube nommé...) `fic` **avec écrasement** éventuel
- `cmd 2>> fic` : redirection de la sortie erreur standard de la commande `cmd` sur le fichier (ordinaire, tube nommé...) `fic` **en concaténation**
- `cmd1 | cmd2` : redirection de la sortie standard de `cmd1` et de l'entrée standard de `cmd2` sur un même tube anonyme
- `cmd <( cmd1 ) ... <( cmdn )` (*substitution de processus*) : redirection de la sortie standard de chaque commande `cmd1` ... `cmdn` vers un flot dont la référence est utilisée comme paramètre de la commande `cmd`.

Les espaces de part et d'autre des symboles de redirection sont requis.

Par ailleurs, dans un *pipeline* `cmd1 | cmd2 | ... | cmdn`, les redirections additionnelles sont autorisées :

- redirection de l'entrée standard de `cmd1`
- redirection de la sortie standard de `cmdn`
- redirection des sorties erreurs des `n` commandes

En cas d'échec d'une redirection, la ligne de commande saisie n'est pas exécutée, et la valeur de retour est 1.

Gestion des signaux

`jsh` ignore les signaux `SIGINT`, `SIGTERM`, `SIGTTIN`, `SIGQUIT`, `SIGTTOU` et `SIGTSTP`, contrairement aux processus exécutant des commandes externes.

*(optionnel) Si vous le désirez, vous pouvez améliorer le comportement d'`exit` de la manière suivante : si `jsh` a des jobs en cours d'exécution ou suspendus, après un premier appel à `exit` sans effet, un deuxième appel de confirmation, **immédiatement après le premier**, provoque cette fois la terminaison de `jsh`; `jsh` doit*

cependant en informer tous les processus des jobs en cours par l'envoi d'un signal `SIGHUP`, après avoir relancé les évntuels processus stoppés.

Modalités de réalisation (et de test)

Le projet est à faire par équipes de 3 étudiants, exceptionnellement 2. La composition de chaque équipe devra être envoyée par mail à Dominique Poulalhon avec pour sujet `[SY5] équipe projet` **au plus tard le 20 novembre 2023**, avec copie à chaque membre de l'équipe.

Chaque équipe doit créer un dépôt `git` **privé** sur le gitlab de l'UFR **dès la constitution de l'équipe** et y donner accès en tant que `Reporter` à tous les enseignants du cours de Système : Raphaël Cosson, Isabelle Fagnot, Guillaume Geoffroy, Anne Micheli et Dominique Poulalhon. Le dépôt devra contenir un fichier `AUTHORS.md` donnant la liste des membres de l'équipe (un par ligne, en précisant pour chacun, dans l'ordre : nom, prénom, numéro étudiant et pseudo(s) sur le gitlab).

En plus du code source de votre programme, vous devez fournir un `Makefile` tel que :

- l'exécution de `make` à la racine du dépôt crée (dans ce même répertoire) l'exécutable `jsh`,
- `make clean` efface tous les fichiers compilés,

ainsi qu'un fichier `ARCHITECTURE.md` expliquant la stratégie adoptée pour répondre au sujet (notamment l'architecture logicielle, les structures de données et les algorithmes implémentés).

Le projet doit s'exécuter correctement sur lulu.

Il n'est pas exigé que le projet soit portable sur d'autres systèmes. Certaines fonctionnalités demandées (`jobs -t` et la substitution de processus, au moins) sont en effet beaucoup plus simples à réaliser sous linux grâce au pseudo-SGF `/proc`.

Les seules interdictions strictes sont les suivantes : plagiat (d'un autre projet ou d'une source extérieure à la licence), utilisation de la fonction `system` de la `stdlib` et des fonctions de `stdio.h` manipulant le type `FILE` pour les redirections.

Pour rappel, l'emprunt de code sans citer sa source est un plagiat. L'emprunt de code en citant sa source est autorisé, mais bien sûr vous n'êtes notés que sur ce que vous avez effectivement produit. Donc si vous copiez l'intégralité de votre projet en le spécifiant clairement, vous aurez quand même 0 (mais vous éviterez une demande de convocation de la section disciplinaire).

Modalités de rendu

Le projet sera évalué en 3 phases : deux jalons intermédiaires, sans soutenance, et un rendu final avec soutenance. Les deux jalons intermédiaires seront évalués par des tests automatiques.

Premier jalon jeudi 30 novembre

La version à évaluer devra être spécifiée à l'aide du tag `jalon-1`.

Pour créer le tag : se placer sur le commit à étiqueter et faire :

```
git tag jalon-1
git push origin --tags
```

Points testés :

- existence du dépôt git
- présence du fichier `AUTHORS.md`
- compilation sans erreur avec `make` à la racine du dépôt
- exécution de `jsh` à la racine du dépôt
- exécution des commandes externes simples à l'avant-plan (avec arguments mais sans redirection ni possibilité de basculer à l'arrière-plan)
- bon fonctionnement de `cd`, `pwd`, `?`, `exit` (en l'absence de jobs)
- conformité du prompt (en l'absence de jobs)

Deuxième jalon jeudi 21 décembre

La version à évaluer devra être spécifiée à l'aide du tag `jalon-2`.

Points testés : ceux du premier jalon, plus :

- exécution des commandes externes simples à l'arrière-plan avec `&` (sans possibilité de basculer à l'avant-plan, et seulement pour les commandes qui ne cherchent pas à lire ou écrire sur le terminal)
- bon fonctionnement de `jobs` (sans option), `kill` et `exit`
- adaptation du prompt selon le nombre de jobs en cours
- redirections autres que `|` et `<( )`

Rendu final début janvier

La version à évaluer devra être spécifiée à l'aide du tag `rendu-final`.

Le projet final devra être rendu à une date encore à définir mais au plus tard le 12 janvier, pour des soutenances entre le 15 et le 19 janvier.

Participation effective

Les projets sont des travaux de groupe, pas des travaux à répartir entre vous ("je fais le projet de Prog fonctionnelle, tu fais celui de Systèmes"). La participation effective d'un étudiant au projet de son groupe sera évaluée grâce, entre autres, aux statistiques du dépôt git et aux réponses aux questions pendant la soutenance, et les notes des étudiants d'un même groupe pourront être modulées en cas de participation déséquilibrée.

En particulier, un étudiant n'ayant aucun commit sur les aspects réellement "système" du code et incapable de répondre de manière satisfaisante sera considéré comme défaillant (DEF).