Projet : Programmation Système

Lilian CHAMPROY, Kinda AL CHAHID
Axel DUBROCA

9 janvier 2016

Table des matières

Chapitre 1

Juste un exemple

C'est pas mon truc définitif. C'est juste histoire que moi, j'ai une trace à la fin. Je rédigerai tout correctement à la fin.
Retour simple.

> Initialisation de l'ornithorynque

Dépôt pour la PRS sur mon GitHub. C'est "Party in the SSH". Ya tous les fichiers de base. Notre mission pour l'instant ? Il faut réfléchir à comment faire ce Remote Shell. Normalement, presque tout le reste sera bon, puisque je l'avais faite l'année dernière. Je ne l'ai pas encore fait, je réfléchis au Remote Shell cette après-midi. Éventuellement, je m'occupe du mini-shell ce soir.

Ce que j'entends par "réfléchir au Remote Shell", c'est réfléchir à, techniquement, comment mettre en place tout ça. Pas besoin de code, des dessins suffisent (et pourquoi pas du pseudo-code).

Voilà, n'oubliez pas qu'il y a un rapport final à faire, donc prenez des notes sur absolument tout ! Le rapport final sera découpé en chapitres. Exemple :

1/ <titre1> (<auteur1>)
2/ <titre2> (<auteur2>)
...

Ainsi de suite. C'est faisable facilement en LaTeX, avec des import ou un truc du genre. M'en souviens plus. Lilian tu me confirmeras. On aura le fichier Rapport.tex à compiler à la racine, puis un dossier chapters dans lequel on mettra les chapitres, qu'on nommera "<auteur>-01", "<auteur>-02", et la numérotation est personnelle (chacun aura son 01, 02... on triera à la fin l'ordre final). Avantage de faire comme ça ? Tout les trucs compliqués de LaTeX seront dans le compilable, et tout le reste sera plus simple. Faut pas hésiter à mettre des images, dessins, plein de bordel pour montrer qu'on gère. Par contre, c'est 15 pages max. Donc on se donne 5 pages A4 chacun environ (juste un ordre d'idée, ça dépendra ensuite).

Et n'oubliez pas, il y a une grille d'auto-évaluation à la fin du Sujet. On a un aperçu de la difficulté de chacune des parties, et surtout, de la proportion de chacun dans la note finale. Même si le Remote Shell (par exemple) n'est pas fonctionnel, il ne sera pas si pénalisant.

> Manuscrit 1

Pour l'IDE, je prends Code : :Blocks. Pas forcément le mieux (et clairement pas un choix de coeur tellement je le trouve laid), mais habituellement, pour du C, je ne prends pas de véritable IDE (Sublime Text *tousse*), donc j'ai envie de changement. Le Makefile est fourni, et le programme s'éxécute dans le terminal, donc je n'ai pas de réel besoin des fonctionnalités de ce programme.

Concernant le Remote Shell, l'idée actuelle est de créer un processus par connexion SSH, qui exécutera les commandes qu'il reçoit dans son STDIN. La question étant : est-ce faisable ? La ligne de commande n'est pas habituellement donnée via STDIN.
La solution qui me semble sous-entendue dans le sujet est d'exécuter à chaque fois "ssh host cmd". Ça me paraît vraiment lourd, et ça ne conserve pas les connexions. De même, comment ouvrir une connexion ssh avec mot de passe au travers de notre processus principal ? On peut rediger l'entrée vers la sortie, mais

comment savoir à quel moment redonner la main à notre mini-shell ?

La commande mkfifo semble intéressante pour cela. Elle crée une file nous permettant d'écrire dans le STDIN d'un processus : "mkfifo myfifo ; ssh host < myfifo ; echo lol > myfifo" permet d'envoyer "lol" dans le STDIN du processus SSH. Ça ressemble grandement à ce que je veux ! Actuellement, ma connexion Internet est prise par les innombrables mises à jour (joie), donc ssh met trop de temps à répondre. Je testerai plus tard. Mais si ça fonctionne, c'est probablement ce qu'on utilisera ! Le seul problème que je peux y voir actuellement, c'est comment créer et conserver notre file. À voir plus tard. De même, l'utilisation d'une file semble aspirer le STDOUT du programme...

J'avais oublié à quel point il est agréable de créer un projet versionné via git, et géré par Code : :Blocks. Ce sera Sublime Text. Pas particulièrement fonctionnel, mais je m'en sortirai avec ça.

Je vais tester en ligne de commande si écrire dans STDIN fonctionnerait.
$> mkfifo fifolol
mkfifo : impossible de créer la FIFO «fifolol» : Le fichier existe

Oh ? Je pensais que ces files étaient temporaires. J'avais déjà donné ce nom à ma file précédente, quand j'ai découvert cette commande. Effectivement, en lançant ls, je vois que mkfifo crée un fichier fifolol. Je peux le supprimer avec rm. C'est une sorte de "faux pipe" si je puis dire.

$> bash < fifolol &
$> cd /
$> echo "pwd" > fifolol
bash : fifolol : Permission non accordée

Oh ? Encore une erreur ? Je n'ai pas les droits sur mon pipe ? Qu'à cela ne tienne : supprimons-le, et recréons-le.
$> mkfifo fifolol
$> bash < fifolol &
$> cd /
$> echo "pwd" > fifolol
bash : fifolol : Permission non accordée

MAIS QUEL CON ! Je fais un cd, puis je nomme fifolol... Qui n'est donc plus là. Quel idiot.
$> echo "pwd" > /fifolol
/home/seiken

MAGNIFIQUE ! ÇA FONCTIONNE ! Bon, le bash s'est terminé une fois la commande exécutée. Mais on peut outrepasser ce problème avec tail :
$> tail -f fifolol | bash &

Là, je peux envoyer des commandes à bash à l'infini. C'est juste. TROP. GÉNIAL. J'ai presque envie d'arrêter le projet ici, tellement je suis refait. Mais bon. Testons maintenant avec un ssh. Je lance la connexion, puis je tente de lui passer une entrée via un autre terminal.
$> cd -
$> ssh adubroca@jaguar.emi.u-bordeaux.fr < fifolol &
(Deuxième terminal : $> echo " *** " > fifolol)
Pseudo-terminal will not be allocated because stdin is not a terminal.

Hum. Embêtant. J'ai tout simplement essayé d'envoyer mon mot de passe via la file (procédé hautement sécurisé, vous en conviendrez. Je l'ai au moins censuré dans ce rapport). Mais le seul moyen de l'entrer, c'est un terminal. Part-on du principe que nous n'avons pas de mot de passe à entrer ? Après tout, au CREMI, puisque nous sommes authentifiés, nous n'en avons peut-être pas besoin. Mais cela rendrait ce projet futile, à mon avis. Bref.

Pour nous changer les idées, occupons-nous de traiter et exécuter l'arbre syntaxique généré par yyparse. Compilons et exécutons le code de base, voir ce qu'il fait déjà.

Wow. Erreur fatale : le fichier readline.h n'existe pas. J'avais oublié que je fais partie de ceux qui n'ont visiblement même pas les librairies standards (du genre, pas de floor ni de sqrt dans mon math.h). J'ai

toujours une chance incroyable avec mes OS depuis mon dual-boot (vous avez saisi l'ironie). Il semblerait que ce soit grub qui m'embête. Et ça, c'est du au changement de disque dur... Bordel.

Erreur suivante : yacc. Il est vrai que je n'en ai jamais eu besoin avant. Plus qu'une erreur : "ne peut trouver -ly". Tout a décidé de m'embêter aujourd'hui. Heureusement que je suis doublant, je sais que -ly est lié à bison. Pas certain que ce soit une information facile à trouver.

C'est bon, le mini-shell est en route. Je constate avec plaisir que le système d'historique est déjà implémenté. Soyons honnêtes : l'année dernière, c'était la partie la plus difficile du projet. Alterner entre modes raw et cooked du terminal... Je ne sais pas si c'était la bonne solution, mais c'était pas mignon. L'arbre a l'air inchangé par rapport à l'année dernière. Vous savez ce que cela veut dire... Je vais pouvoir reprendre une bonne partie du code précédent. Il était bon, seuls les appels aux fonctions données et éventuellement les valeurs de retours vont changer. Je pourrai aussi revenir sur les redirections et les pipes. Il me semble me souvenir que nous avions des solutions fonctionnelles, mais pas non plus exceptionnelles.

... Je comprends mieux pourquoi l'arbre est similaire. Le code source est identique (à my_yyparse et main près). Sans surprise, après tout. La base du sujet est la même. Le seul véritable changement, c'est le traitement des commandes "internes" qui est dans un autre module. Je ne vais pas pour autant simplement copier / coller. Le code précédent n'était pas correctement commenté, chose que je vais changer maintenant.

Note pour plus tard : J'ai remarqué que la grille d'auto-évaluation faisait référence à des processus zombies. Pour éviter cela, on mettra en place une structure de données (tableau, liste chaînée) contenant les pids de tous les processus qu'on lancera. Quand notre programme se termine, on peut donc savoir quels processus sont toujours actifs, et choisir quoi faire.

Hum. Je viens de refaire la fonction d'affichage de l'arbre. J'avais commencé à l'écrire dans Shell.c, puis j'ai voulu la tester (un modèle basique), puis j'ai vu que c'était déjà inclus... J'avais oublié. J'avais en plus fait la remarque plus tôt. Idiot. Du coup, je l'ai pas refaite. Je changerai juste celle déjà donnée, je suis pas fan de l'aspect final. Ou je la referai. Non, je la referai. Je préfère.

Emacs. I'm back. Je ne sais pas pourquoi, mais l'indentation déconne complètement sous Sublime Text. C'est n'importe quoi. Puisque je ne nécessite aucune fonctionnalité particulière, je risque de finir sous Gedit. Il y a la coloration syntaxique, c'est tout ce dont j'ai besoin.

Fate exists. And it doesn't like me. Je n'ai plus le code de l'année dernière. En tous cas, pas dans son intégralité. J'ai, pour une raison abscons, une ancienne version du projet. Embêtant. Mais rien de grave. Je peux le refaire.

Premier problème. Je suis au switch / case principal, qui choisit la routine à appliquer selon le type de noeud. SIMPLE, SEQUENCE, SEQUENCE_ET et SEQUENCE_OU ne posent aucun problème. Par contre, BG, une fois appliqué, fait planter la suite du mini-shell. La lecture en entrée n'est pas forcément correcte, l'affichage saute une fois sur deux (un "\n" qui disparaît)... Drôle de comportement. Je continue avec les redirections, voir si le problème peut affecter d'autres parties, pour éventuellement trouver un lien.

La redirection d'entrée fonctionne partiellement : une fois la commande ayant son entrée redirigée exécutée, le mini-shell se termine.

JE SAIS D'OÙ VIENNENT CES ERREURS. Encore une erreur stupide. J'oublie les flags O_RDWR / O_RDONLY / O_WRONLY. Forcément, les processus n'aiment pas trop. Il ne faut pas non plus que j'oublie de réinitialiser les entrées / sorties à la fin de l'exécution (je dis ça car je viens de me faire avoir). Tout fontionne bien mieux maintenant ! Les bugs étaient bien dûs à ça, et aux redirections. Une petite question demeure. Pour STDERR, dois-je utiliser O_TRUNC ou O_APPEND ? Instinctivement, je dirais O_TRUNC. Mais je n'en suis pas certain. Les exemples que je trouve sur Internet mettent les deux... C'est moche. Well... Let's give it a try.

C'est pas encore ça. La commande pipée s'exécute, mais il semblerait que STDIN ne se restaure pas, comme si le mini-shell attendait encore quelque chose sur son entrée. Mais écrire ne fait rien. C'est étrange, puisque les redirections de STDIN et STDOUT fonctionnent indépendamment. Ici, ce n'est qu'une combinaison des deux. Qu'est-ce qui fait que le processus attende toujours une entrée ? Bien que

j'ai une idée. EEEETTTT... C'ÉTAIT BIEN ÇA ! Il suffisait que je ferme le pipe. La deuxième commande attendait que le programme principal, seul processus restant avec le pipe ouvert, écrive. C'est intéressant, car ça montre un aspect des pipes que je ne connaissais pas. Je pensais que, puisque plus aucun processus n'écrira dans le pipe, il était considéré comme inutile, et que le processus à l'autre bout n'attendrait pas. Preuve que je me trompais.

Argh, j'avais oublié à quel point je n'aime pas le LaTeX... J'en comprend l'intérêt, mais... Non. J'aime pas ne pas avoir le contrôle absolu de mes fichiers. Anyway.

> Gonna execute'em all !

Il est temps de s'occuper des commandes internes. La stratégie ? On va avoir une liste de ces commandes, dans le cas d'une commande SIMPLE, on compare la commande dans l'arbre avec les éléments de notre liste, on récupère son indice et on switch / case. Rien de compliqué. On aura ensuite une fonction par commande. À la fin de notre liste, j'ajoute un NULL pour détecter facilement la fin (tant que non NULL...). D'ailleurs, erreur que j'ai commise, c'est de faire if(strcmp(...)). strcmp retourne 0 en cas d'égalité, et 0... veut dire false. L'habitude qu'un programme retourne 0 en cas de succès, et un code d'erreur sinon. Du coup, forcément, mes commandes internes n'étaient pas bien traitées. Pour les commandes internes, par ailleurs, on ne traitera que les cas les plus standards. Je suppose que l'objectif n'est aps de les coder entièrement (c'est plus long que difficile). Par exemple, pour la commande kill, on ne traitera que le cas "kill <pid1> <pid2>...". Faisons une liste des traitements :

echo : afficher les arguments sur la sortie standard ;

date : en utilisant time() et localtime(), on peut obtenir toutes les informations dont on a besoin, puis l'afficher correctement. Pour l'instant on reste sur un affichage simple, pas d'options comme la commande d'origine. Pour avoir le fuseau horaire, on compare le localtime() avec le gmtime(), et ensuite ce n'est que de la mise en forme ;

exit : je m'en occupe maintenant, car j'en ai marre de quitter le mini-shell avec un C-c. C'est simplement un exit(0) ;

pwd : on va utiliser getcwd(NULL, 0) qui a l'avantage d'utiliser malloc, et donc de ne pas avoir de limite de taille de buffer (même si 1024 serait certainement suffisant). Il faut juste qu'on free() ce buffer juste après l'affichage ;

cd : il semblerait de la fonction chdir() fasse le boulot. Seuls "problèmes" : "cd" seul est équivalent à "cd ~", et "cd -" doit revenir en arrière. Oh ! "cd ~" renvoie un Syntax error. Bon, pas besoin de le faire du coup, je suppose (on va pas modifier l'analyseur quand même, si ?). Du coup, je suppose de même avec "cd -" (il faudrait juste stocker le buffer donné par pwd, et s'en resservir dans chdir) ;

history : le programme possède déjà un système d'historique (haut et bas permettent de se déplacer dedans). Toutes les commandes sont donc forcément enregistrées quelque part ; il suffit de les trouver, et de les afficher. L'historique semble être une liste de HIST_ENTRY, pointeur sur une structure contenant une ligne, qui doit être la commande. On devrait donc utiliser les fonctions history_get() qui nous renvoie une de ces structures, et where_history() pour savoir jusqu'où nous devons afficher l'historique (position actuelle). Pour l'affichage, je conserve ce qui se fait avec la commande history habituellement : 5 colonnes pour le nombre, deux espaces, puis la commande. La commande de base s'adapte sûrement à la largeur des nombres, mais ici, il est peu probable qu'on dépasse les 99'999 commandes exécutées ;

hostname : il existe la fonction gethostname() qui fait le job. On lui donne un buffer de 1024 caractères (qui devraient suffire), et on ajoute, au cas-où, un '\0' à la fin, car si le nom était trop long et a été tronqué, le '\0' de fin est tronqué aussi ;

kill : on a la fonction kill(). Le signal par défaut utilisé est SIGTERM, signal 15. On vérifie que tous les paramètres sont des nombres, puis on applique kill() sur chacun d'eux.

Voilà pour les commandes internes. À ce stade, il nous reste "remote", qui s'annonce quelque peu tiré par les cheveux.

> Get inside of the party !