Termina

Projet de Programmation Système

CHAMPROY Lilian
AL CHAHID Kinda
DUBROCA Axel

Licence 3 Informatique
Groupe IN501A1
Chargé de TD : M. WACRENIER Pierre-André
Janvier 2016

FIGURE 1 – Tribute to Miley Cyrus, who gave us a taste of bool chained lists.
(Bon, ok, ça se dit linked lists...)

Table des matières

Chapitre 1

Initialisation du projet

> Initialisation de l'ornithorynque [1]

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.

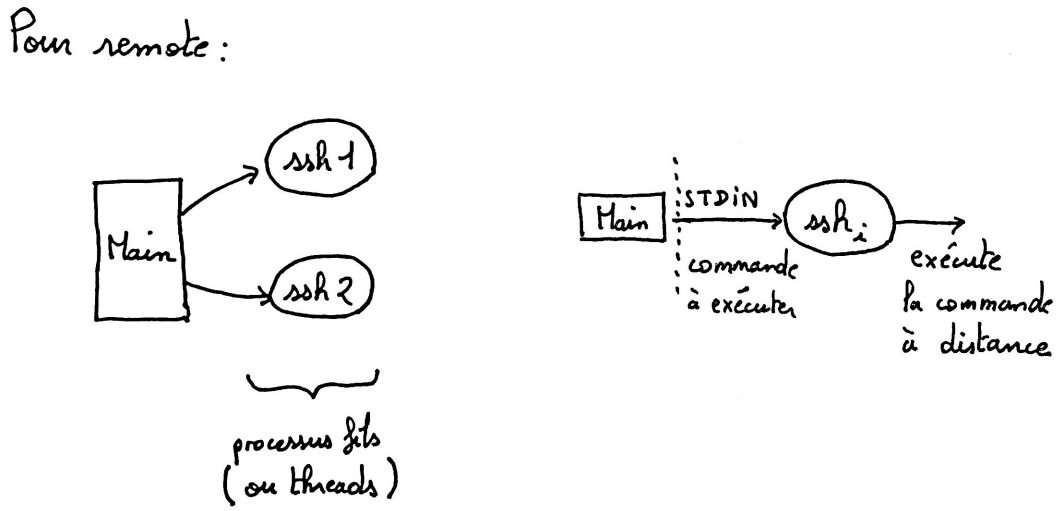
1. Il s'agit ici du nom d'un commit. Ils sont indiqués, pour que vous puissiez vous repérer dans le temps.

Chapitre 2

Première réflexion sur le remote shell

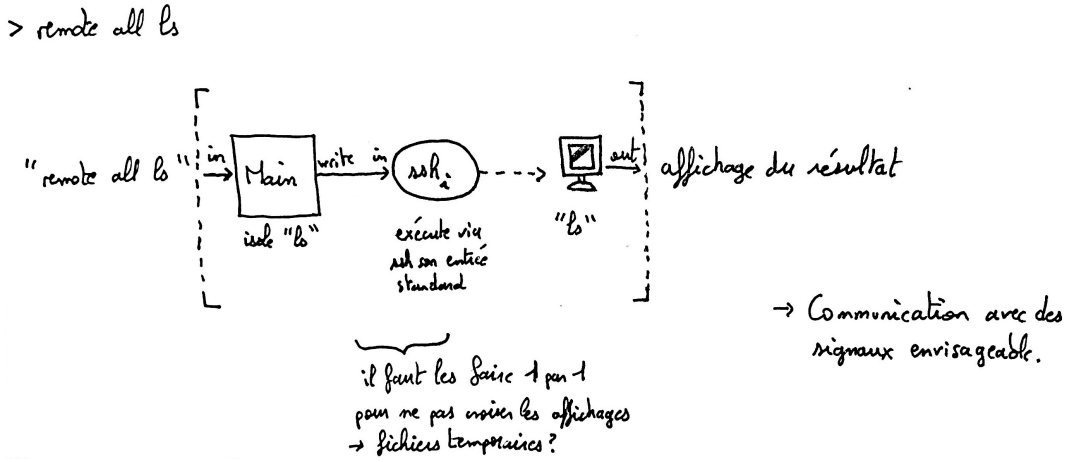
Pour l'IDE, je prends Code : :Blocks. Pas forcément le mieux (et clairement pas un choix de coeur tellement je le trouve laid), mais habituellement, pour du C, je ne prends pas de véritable IDE (Sublime Text *tousse*), donc j'ai envie de changement. Le Makefile est fourni, et le programme s'éxécute dans 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 est-elle habituellement donnée via STDIN ?




```
> remote add m1 m2 ...
  while ( argv[2] ) {
      fork {
          exec ("ssh"+argv[2]);
      }
      shift;
  }
→ enregistrer les pids / noms
```

```
>remote remove
  foreach ( proc in list ) {
      write (proc, "exit", 4);
  }
```



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 rediriger 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.

Chapitre 3

Implémenter l'évaluation des commandes

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.

En fait non, cette ancienne version est vraiment incomplète. Je n'ai qu'une partie du switch / case de la fonction de parcours de l'arbre syntaxique. Je sais pas ce que j'ai fait du reste... Certainement perdu lors d'une mise à niveau, je n'avais pas du sauvegarder mes données. Je suppose. Enfin bref.

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 !

Chapitre 4

Traiter les commandes internes

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 pas 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 !

Chapitre 5

Shell → Termina

Pour le plaisir, on se fait une commande Easter-Egg. Notre programme final va s'appeler Termina (sans le "l", c'est volontaire). Termina est le nom de la région du jeu "The Legend of Zelda : Majora's Mask". La commande (interne) permettant de lancer cet Easter-Egg est donc... "majora". Allez au prochain paragraphe si vous ne voulez pas être spoilés. La commande affiche un compteur (qui augmente) comptant le nombre de jours et d'heures passées depuis la date de rendu, le lundi 11 janvier, à minuit. Bon, j'avoue n'avoir implémenté qu'un calcul simplifié... Dès qu'on changera de mois, ça ne fonctionnera plus. Mais je vais pas batailler pour ça. Ça n'a pas pour objectif d'être pérenne.

Plus sérieusement, je repasse sur le code afin de commenter un peu mieux, et il faut que je modifie le Makefile (changer le nom du programme) et le Readme (indiquer comment exécuter le programme). Une fois fait, je fais un dernier commit avant d'attaquer le gros morceau : "remote". J'ai d'ailleurs enfin utilisé le fichier Evaluation.c. J'ai laissé tout le code fourni de base dans Shell.c, et déplacé tous les ajouts dans Evaluation.c.

> You've met with a terrible fate, haven't you ?

Chapitre 6

Le remote shell

Retour au remote shell. Comme il avait été testé, le processus ssh ne prend pas ce qu'on lui envoie depuis la file car ce n'est pas un terminal. La question étant : qu'est-ce qu'il concerne comme un terminal ? Si le programme crée un processus bash qui crée le ssh, est-ce que cela lui irait ? Testons cela.

Oh, juste avant de le tester, j'ai ajouté au programme une option, utilisable via "-v" ou "–verbose". Si elle est activée, le programme affiche l'arbre syntaxique avant d'exécuter la commande. Rien d'incroyable.

Bon, je sais, je temporise beaucoup, mais je viens d'effectuer des tests pour le mode verbeux, et j'en ai profité pour lancer notre programme dans notre programme. Aucun souci. Mais je me dis que si le remote shell fonctionne (si on y arrive), on nous demandera probablement d'exécuter "remote all ./Termina". Et ça doit fonctionner. Je m'inquiète un peu pour cette fonctionnalité.

Je viens de regarder le scipt shell xcat.sh. Constat : il y a la commande mkfifo. Maintenant, ne serait-il pas plus simple d'ouvrir des fenêtres, et de s'en servir comme entrée / sortie de nos connexions ssh ? Et que le tout reste contrôlable via notre terminal principal ?

AH ! J'AI OUBLIÉ LES PROCESSUS ZOMBIES ! Shit. Bon tant pis, ce sera pas fait. Il fallait juste créer uns structure de données du genre liste chaînée ou tableau extensible, et enregistrer les pids à chaque fork. Dès qu'un wait avait fini, on retirait le pid de la liste. En fin de programme (exit, éventuellement intercepter quelques signaux), on parcours la liste et on les tue tous à grand coup de "kill -9".

J'essaie, actuellement, de simplement lancer un fils ssh. Aucun affichage ne se fait sur stdout. J'ai bien peur que ssh n'accepte réellement que les terminaux en entrée. Je fais quelques recherches sur le net, mais ça n'a vraiment pas l'air simple. Il semblerait que ce soit cette seule phase d'authentification qui bloque tout. La suite n'a pas l'air difficile, ça n'a l'air d'être que de l'écriture dans un pipe.
Histoire de suivre mes recherches, je vais lister toutes les solutions que je vois passer, sans qu'elles soient forcément adaptées (certaines n'ont même rien à voir). J'ai aussi demandé à certains ce qu'ils avaient fait, sans détailler. Deux réponses : "ssh <serveur> bash", et "libssh". Le premier, ça ne laisse pas passer l'authentification et mess-up avec les entrées / sorties, et le second, c'est hors de question d'utiliser une bibliothèque, c'est un choix trop facile. Bref, voilà la liste :

- screen
- nohup
- disown
- xterm/gnome-terminal
- Cluster SSH
- PAC Manager
- OpenSSH

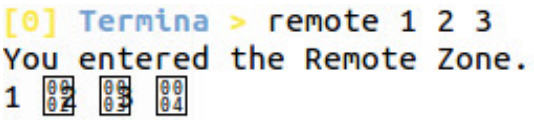
Tiens, un tutoriel pour le faire en Python [1]. Et devinez la première phrase : "Using subprocesses to interact with SSH is notoriously difficult". No kidding. "It is recommended that you just ssh-copy-id to copy your public key to the server so you don't need to enter your password, but for the purposes of this demonstration, we try to enter a password".
J'ai donc la confirmation que je voulais. Ce projet demande sûrement de se connecter à des serveurs sans authentification. Cependant... Cependant je ne me réduirai pas à ça. Cette commande est juste incroyable.

1. https://amoffat.github.io/sh/tutorials/2-interacting_with_processes.html

Si on parvient à la mettre en place sous cette forme, c'est beaucoup mieux. Contrôler pleinement depuis un seul terminal plusieurs connexions ssh. Certes, je désobéis au sujet, mais "remote" n'a plus aucun intérêt si on zappe l'authentification. Le projet ne sera probablement pas terminé, car il me reste peu de temps, mais je veux rester là-dessus.

Je lis donc le tutoriel que j'ai mentionné plus tôt. Et il parle de buffering ! Le problème n'est pas que l'entrée n'est pas un terminal, mais que le buffer contient un '\n'. En changeant la méthode de bufferisation, on peut régler ce problème. Dans le sujet, ils mentionnent fdopen() et setvbuf() pour cet effet. J'ai trouvé des exemples d'utilisation, mais pas sur des pipes. J'ai du mal à comprendre la façon de l'utiliser. Je crée un seul fichier FILE* que je bufferise, et les deux processus lisent / écrivent dedans (une sorte de faux pipe), ou je dois créer mon pipe, créer les FILE* correspondant aux deux extrémités, et les bufferiser ? C'est simple : aucun des deux ne fonctionne. Je ne peux pas rediriger des entrées sorties dans des flux, dup() ne fonctionne qu'avec des descripteurs de fichiers.
La solution privilégiée par le tutoriel, c'est de ssh-copy-id avant tout. Pour notre programme, ça voudrait dire que si la connexion ne se fait pas, on demande à l'utilisateur de lancer ssh-copy-id en premier. On revient à la connexion sans identification. Juste pour une histoire de bufferisation.

J'y vais progressivement. J'ai trouvé la réponse à ma question. Le côté lecture du pipe, on s'en fiche. C'est l'écriture qu'on prend comme stream. En faisant ça, on décide quand est-ce que le pipe reçoit de l'information. Je dois maintenant utiliser fputs() pour écrire dans mon stream / pipe. Ça fonctionne assez bizarrement tout de même, mais ça m'arrange. Si je dois envoyer la commande au ssh, je crois l'envoyer d'un bloc. Je suppose. Donc je dois faire plusieurs fputs(). Mais après un test (afficher une chaîne spéciale après l'affichage du buffer par le fils), plusieurs fputs() à la suite ne déclenchent pas de flush du stream. Ce qui est parfait pour envoyer la commande mot par mot, et ajouter les espaces, sans batailler.
Hum. Ce n'est pas parfait. L'ajout des espaces semble créer un décalage de bits. Des caractères étranges s'affichent.



Je ne sais trop peu quoi en penser. Je tente quelque chose... Et ça fonctionne. Je passais un caractère espace par référence, et ça déconnait, alors qu'en l'écrivant en dur dans fputs (je ne savais pas qu'on pouvait... c'était logique pourtant), ça passe sans souci.
Bien ! On arrive a envoyer des donnés via un buffer. Maintenant, il faut tester d'envoyer un mot de passe à une connexion ssh via ce buffer, et tester différentes bufferisations. Pour l'instant, ssh affiche bien sur le terminal, et demande le mot de passe. Problème : je ne peux pas le valider. Là, j'étais en _IONBF (sans bufferisation. Tentons par ligne, _IOLBF. Pas mieux. Et en bufferisation totale, _IOFBF ? Toujours pas. Je pensais que bufferisé par ligne fonctionnerait. Et si je tente d'afficher quelque chose une fois que le père a fini ? AAAAHHHH, JE NE COMPRENDS RIEN ! Avec fwrite(), c'est le même délire. Je ne comprends pas. Le comportement change. Selon quand j'appuie sur Enter, le programme ne fait pas la même chose. Des fois, il est ajouté au buffer, parfois affiché, parfois on passe au read() suivant. Sans bufferisaton, je comprends mieux. Enter provoque un flush à chaque fois.

J'ai possiblement un indice sur le problème. J'ai essayé de ne pas m'occuper du fils, et de tout simplement afficher le buffer une fois lu, et envoyé dans le stream. Il est lu, et bien affiché. Le père boucle à l'infini dans le while(read). Rien de surprenant. La sortie, que ce soit du père ou du fils, est conforme à l'entrée, incluant le '\n'. Ça ne vient pas de la communication, mais encore du ssh... Quel bordel.

Ok. C'est bon. Je suis vaincu. J'abandonne. J'y arrive pas. Ça fait plusieurs heures que je suis dessus, et aucune avancée. J'en peux plus. Je n'ai jamais autant eu cette impression d'échec. Je n'avance sur rien.

Chapitre 7

The Remote shell mission :

FAILED

C'est un échec. Alors que je me bats depuis des heures avec le processus ssh, il a gagné (Si j'avais pu faire un dessin d'un processus lourdement armé, je l'aurais mis ici).

Cependant il y a quelques directions que j'aurais pu prendre :

- libssh : c'est une solution utilisée par d'autres étudiants. Si le moral revient après ma défaite, j'explorerai cette possibilité (même si je considère tout de même qu'il s'agit, en quelque sorte, d'un court-circuit du projet) ;
- ssh <serv> <cmd> : s'il n'y a pas d'authentification, je peux lancer les commandes isolées. C'est pas très propre, puisqu'il faut relancer la connexion en permanence ;
- ssh <serv> bash : encore une solution utilisée par d'autres étudiants. C'est une sorte d'amélioration de la précédente. Elle a l'avantage de conserver la connexion. Elle outrepasse aussi un problème : elle permet de continuer à interagir avec notre processus. Après un exec(), à moins qu'il y ait une erreur, on ne peut plus rien faire.

Je pense qu'une fois l'authentification passée, le mieux est de lier un terminal à chaque processus, que ce soit en entrée ou en sortie. Le programme principal conserve un droit d'écriture dans l'entrée standard de ces processus ssh. Ça a l'avantage d'agir comme si chaque terminal s'était connecté indépendamment, mais le tout lancé via un seul, et aussi de pouvoir tout de même lancer des commandes via le principal, voir même lancer des commandes à tous (on pourrait sans difficulté créer une commande "remote send <serv1> <serv2>..." qui demande ensuite une commande, et l'envoie à ces serveurs précis). Il y aurait moyen de faire des choses sympas, et pratiques.
Je regarde libssh, et ça simplifie énormément la chose. Dans un sens, traiter une commande interne avec des fonctions C, c'est plus logique, mais bon. C'est avec cette librairie qu'il fallait sûrement partir. Même les autres solutions me semblent trop archaïques pour que ce soient celles attendues. L'authentification est de plus possible avec libssh. Il suffit de le demander sur le terminal, sans l'afficher, et de l'envoyer à ssh_userauth_password().

> Failure

Quelques ajustements avant de procéder au dernier push de ce projet. Je confirme que remote ne sera pas fait. Les #include inutiles ont été retirés. Vous trouverez d'ailleurs sur la page suivante un schéma de dépendance des fichiers locaux.
C'est ici que ce terminent ce projet, et ce rapport.

> The End

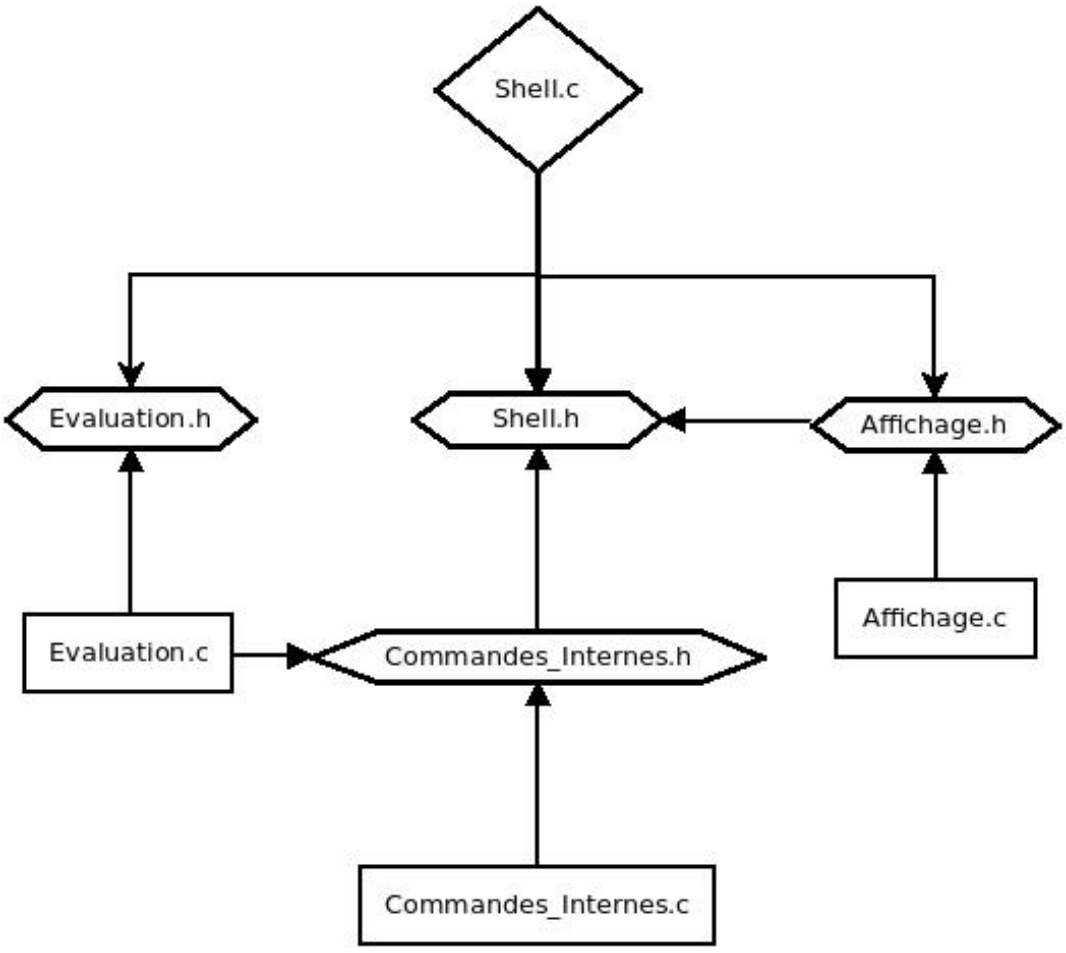


FIGURE 7.1 – Schéma de dépendance des fichiers locaux de ce projet