Rapport de Projet “testaro”
Réseaux & Systèmes

Bouguessa Amandine - Zaghrini Eloïse

Introduction

Ce rapport présente notre travail sur le projet testaro du cours de Réseaux & Systèmes proposé par Télécom Nancy et Mines de Nancy.

Nous avons implémenté un programme testaro permettant de réaliser une série de tests à partir d’un fichier de description pris en argument qui contient les tests à exécuter avec leur résultat attendu.

Extensions réalisées :

- 1. Les lignes ’p’ affichent leur argument sans l’exécuter
- 2. Compter le numéro de ligne pour situer les messages d’erreur dans le fichier.
- 5. ! set timeout <val> pour modifier cette valeur.
- 6. Si on trouve \\ à la fin d’une ligne, il faut considérer qu’il y a un seul \ et ne pas fusionner cette ligne à la suivante

Nombres d’heures passées sur les différentes étapes du projet :
Amandine :

- Conception : 10h
- Implémentation / Tests : 20h
- Rapport : 3h

Eloïse :

- Conception : 5h
- Implémentation / Tests : 20h
- Rapport : 3h

I. Conception et Implémentation

Nous avons fait le choix d'ouvrir le fichier de description en lecture seule car il ne doit pas être permis d'écrire dedans et il nous suffit de lire ses lignes.
On a créé une boucle while permettant de lire chaque ligne du fichier à l'aide de *getline* et qui appelle une fonction de traitement de la ligne : *treatLine*.

1. Lire et interpréter les lignes : *treatLine*

La fonction treatLine permet de différencier les différents types de lignes afin de les analyser en conséquence avant de sortir de la fonction sans rien retourner à l'aide d'un "*return;".*

Le premier test réalisé est celui qui détermine une ligne blanche : nous avons choisi de mettre une boucle for qui compte le nombre d'espace et le compare à la taille de la ligne afin de repérer les lignes blanches qui peuvent aussi bien être des lignes composées uniquement d'espace que celles avec un retour à la ligne.

Que ce soit pour traiter les lignes finissant par '\' ou celles commençant par '<' ou '>', nous avons choisi d'utiliser *strncat,* sur les différentes chaînes de caractère associées, afin de permettre plusieurs lignes d'affilée de même type.

ex: < Toto \
tutu \
fin
< Tata

est un ensemble de lignes autorisées grâce à *strncat* qui va concaténer les différents éléments.
Ces chaînes de caractère sont vidées, en réinitialisant leur premier caractère à '\0' après chaque commande exécutée.

Le deuxième test à réaliser est donc celui du '\' en fin de ligne afin de concaténer dans *buff* la ligne tant qu'elle finit par ce caractère.

Avant de réaliser tout autre test, on modifie la ligne à analyser en concaténant *ligne* à *buff* puis en copiant *buff* dans *ligne* afin de traiter la ligne entière en tenant compte des éléments antérieurs.

Puis on vérifie la nature de cette ligne complète : commande à exécuter, résultat à obtenir,... en comparant les premiers caractères de la ligne à ceux attendus : "# ", "< ", "> ", "$ ",...
Si la nature de la ligne est reconnue on l'exécute puis on sort de la fonction de traitement afin d'analyser la ligne suivante.
Le type de ligne "$ " appelle la fonction *testFunction* qui va exécuter la commande.

Si la nature de la ligne n'appartient pas à celles testées, c'est qu'il y a une erreur de syntaxe et on interrompt donc le processus.

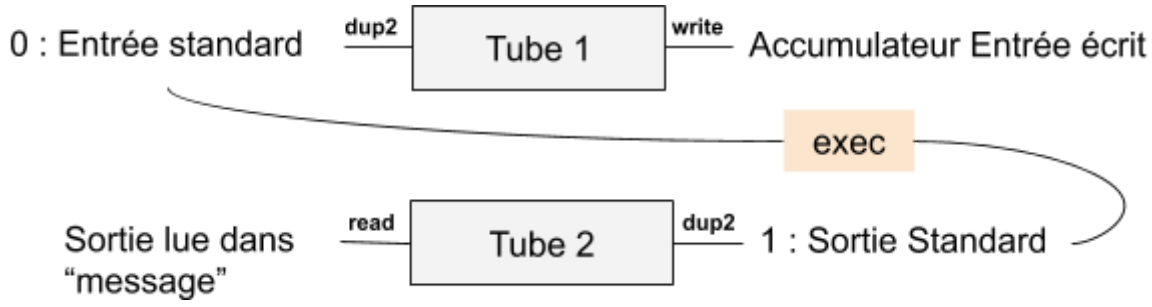
Chaque comparaison de caractère se réalise sur la position de ce caractère. La taille de la ligne et l'écriture exacte dans le fichier de description sont donc des éléments essentiels à la

réalisation des tests. Par exemple, “#Commentaire” ne sera pas reconnu, il faut un espace derrière ‘#’. C’était un de nos choix de ne pas permettre ce type d’écriture.

2. Exécuter une commande : *testFunction*

Pour exécuter la commande demandée par l’utilisateur, nous passons par l’intermédiaire d’un fils, car cela terminerait notre programme autrement. Pour ce faire, nous avons besoin de lui communiquer les éventuels arguments fournis par l’utilisateur désormais stockés dans l’accumulateur entrée, à l’aide d’un tube. Nous avons également besoin de récupérer l’information en sortie d'exécution pour l’analyser par la suite, à l’aide d’un second tube. On utilise deux *dup2* pour relier les entrées et sorties standards aux tubes.

Schéma des tubes et dup2 :



Pour réaliser l'exécution, nous utilisons la fonction execlp en en passant par “sh” avec l’argument “-c” pour ne pas avoir à parser nos arguments. Cette exécution écrit dans la sortie standard et donc dans le tube 2. On lit donc dans ce tube dans le père après exécution et attente du fils. Pour lire ce message dont nous ne connaissons pas la taille, nous lisons par tranche de 1024 caractères à l’aide d’un char intermédiaire “lu”. Nous comparons à l’aide de la fonction *strcmp* le message stocké dans “message” et l’éventuel message d’accumulateur sortie pour vérifier qu’ils sont égaux.

Lorsque le père lance son attente du fils, nous lançons également une alarme de 5 secondes qui représente le délai de garde. Nous avons au préalable modifié à l’aide de sigaction le comportement du programme à la réception de ce signal. Si le père n’a pas de réponse du fils au bout de 5 secondes, le programme se termine.

La commande *cd* est traitée différemment car ne changerait que le directory du fils et non du père pour la suite. Nous utilisons donc la fonction *chdir* directement dans le père lorsque celle-ci est demandée par l’utilisateur.

3. Codes d’erreurs et signaux

Chaque *exit* retourné suite à une erreur est associé à des messages explicatifs utilisant *errno*. Nous imprimons dans le canal *stderr* dédié un message d’erreur puis ce numéro (qui est stocké auparavant dans la variable globale *ERREUR* pour s’assurer qu’il ne change pas au moment de l’imprimer). Nous utilisons la fonction perror pour ensuite afficher un message approprié au numéro d’erreur, puis on effectue un *exit()*.

exit(1)
L'utilisateur n'a pas fourni exactement un argument à testaro
Le fichier fournit à un problème dans son ouverture ou lecture

exit(2)
Au moment de comparer la sortie d'une commande et l'accumulateur Sortie si ces deux sont différents.

exit(3)
Renvoyé par la nouvelle fonction de traitement du signal *SIGALRM* lorsque le délai de garde est dépassé.

exit(4)
Renvoyé si l'exécution d'un appel système a renvoyé une valeur d'erreur (en général NULL ou -1). Par exemple : *read, write, fopen, fclose, fork, pipe, sigaction*..

exit(5 - 39)
La réaction de testaro à chaque signal est modifiée dans une boucle au début du code, à l'aide de sigaction. Chaque signal est désormais associé à la fonction *signalreceived_handler* qui modifie la variable globale *SIGNAL* en le numéro de signal reçu. Lorsque le programme se termine (sans autres erreurs auparavant), il termine avec le code *SIGNAL* + 4.
Il y a 4 exceptions : *SIGALRM* est utilisé pour le délai de garde, *SIGKILL* et *SIGSTOP* qui ne sont pas modifiables, et *SIGCHLD* qui par notre choix est également ignoré, car envoyé à chaque fois qu'un fils termine et qui donc à chaque exécution d'une commande.
Nous avions au début choisi d'interpréter la consigne en faisant s'interrompre le programme dès qu'un signal est reçu, mais nous nous sommes rendu compte que cela pourrait mener à des erreurs et cela ne nous semblait pas très logique d'interrompre le programme à n'importe quel signal.

exit(40+)
Lorsque le fils termine avec une erreur, nous récupérons cette erreur à l'aide de *WEXITSTATUS* et terminons le programme en rajoutant 40 à ce code.
Les commandes malloc sont également testées pour plus de sûreté, et renvoient *errno*+40 en cas d'échec.

II. Extensions

1. Les lignes 'p'

Nous avons fait le choix d'interpréter cette consigne dans le sens d'imprimer sur le terminal de l'utilisateur les arguments suivant une ligne commençant par " p ". (Il a été également discuté d'imprimer le contenant de l'accumulateur entrée qui pourrait également correspondre à la consigne sous une autre interprétation)

2. Compter le nombre de lignes

Nous avons choisi de rajouter une variable globale à notre programme qui est incrémentée à chaque lecture d'une nouvelle ligne, et qui est renvoyée dans les différents prints pour signaler où se situe l'erreur.

3. Essaie de traiter la commande su

Nous avons fait une tentative pour traiter la commande "*su*" qui change l'utilisateur du terminal, qui est en commentaire dans notre code. Ce code fonctionne uniquement si l'utilisateur est en mode root, car change d'utilisateur sans demander de mot de passe. Faute de temps, nous n'avons pas trouvé de solutions pour fournir le mot de passe comme le demande la commande "*su*" normalement et changer d'utilisateur sans être en mode root.

4. `! set timeout<val>`

Pour pouvoir changer la valeur de timeout, nous avons défini une variable globale *TIMEOUT* qui vaut le nombre de secondes après lequel on doit déclencher le signal *SIGALRM* à l'appel *alarm(TIMEOUT)*.
Cette variable est initialisée à 5 mais si une ligne du fichier de description contenant "*! set timeout*", ce qui est vérifié à l'aide d'un *strstr,* alors on change cette valeur par celle donnée dans le fichier.

5. \\ en fin de ligne

Afin de différencier les fins de ligne '\' et '\\', on compare les caractères *ligne[size-3]* et *ligne[size-2]* avec '\', si *ligne[size-3]* vaut aussi '\' alors on est dans le cas '\\' sinon d'une ligne '\', avec *size* la taille de la ligne. Nous supposons qu'il y a un retour à la ligne en fin de ligne, d'où ces indices dans la chaîne de caractère. En effet, utiliser '\' sans autre ligne n'a pas d'intérêt.
Dans le cas d'une ligne '\\', on ne concatène pas à buff la ligne et on supprime le deuxième '\' en le remplaçant par '\n' pour conserver le retour à la ligne, et le dernier caractère est remplacé par '\0' pour marquer la fin de la ligne.

III. Difficultées rencontrées

1. Retours à la ligne

Il a fallu gérer les différents types de lignes et notamment différencier les lignes terminant par un '\n' ou non pour avoir les comportements souhaités. Pour la commande cd par exemple, nous n'avions pas pris en compte au premier abord le fait que la ligne se terminait par un '\n', il ne trouvait donc pas le chemin indiqué.
Pour résoudre ce type de problèmes, nous avons débogué notre code grâce à de nombreux printf nous permettant de visualiser ce qu'il se passait.

2. Commande cd

Nous ne comprenions pas au départ le problème qui se posait à exécuter *cd* dans le fils. Finalement, pour comprendre que le chemin souhaité ne serait pas effectif pour les prochaines lignes à analyser si nous le changions dans le fils, nous avons fait différents essais à l'aide de *getwd* pour visualiser le répertoire où nous nous trouvions à différents instants. Après avoir compris le problème, nous avons refait des tests cette fois ci dans le père avec des tests dans le fichier de description demandant de lire dans un fichier situé dans un répertoire défini plus tôt grâce à *chdir*.

3. Code de retour des signaux

Nous avons eu des difficultés à comprendre la consigne autour des signaux. Plusieurs interprétations nous semblaient possibles : Interrompre le programme à la réception d'un signal et renvoyer le bon code, ou simplement modifier le code de retour final lorsqu'un signal est détecté. Nous avons finalement choisi la seconde interprétation qui nous semblait plus censée et plus en adéquation avec les mots du sujet (terminer et non interrompre). Cependant, nous n'avons pas non plus trouvé de moyen de seulement intercepter un signal et modifier son code de retour avant de laisser notre programme réagir normalement à ce signal. Nous sommes obligés de modifier toute l'action du programme à la réception de ce signal, qui nous semblait assez problématique (si un signal doit interrompre notre programme à quelques exceptions près il ne se terminera pas).

4. Compréhension des cas d'échecs

Nous avons également eu des difficultés à comprendre les codes de retours dans certains cas. Par exemple, si un appel système échoue dans un fils, deux choix sont possibles selon la consigne : renvoyer 4 ou renvoyer le code du fils + 40..

5. Compréhension de certaines extensions

La consigne de certaines extensions laissait une part d'interprétation. Par exemple, l'extension des lignes p a suscité des questionnements comme mentionnés plus haut.
Nous avons essayé d'implémenter l'extension ! import <fichier> mais avons finalement abandonné car nous avions des difficultés à comprendre ce qui était réellement demandé. Nous ne savions pas si nous devions copier les éléments d'un fichier dans le fichier de description ou exécuter ses lignes comme si elles faisaient partie intégrante du fichier de description. Finalement, nous n'avons pas eu le temps d'implémenter cette partie qui s'est révélée difficile.

6. Travail à distance

Travailler en binôme à distance s'est révélé plus compliqué que prévu. Il a été plus difficile de s'organiser ensemble, notamment au début. Nous avons finalement réussi à nous répartir des tâches en nous fixant des réunions sur discord pour discuter ensemble de nos difficultés et du travail restant. Même si cela a été déroutant et nous a pris quelque temps à être mis en place, cela a été constructif et nous permet aujourd'hui d'être plus efficace sur le travail d'équipe à distance.