Projet Testaro

RS - Réseaux et Systèmes

Auteurs :
Niels TILCH
Marie-Astrid CHANTELOUP

Responsables académique :
Jean-Philippe EISENBARTH

18 Octobre 2020 - 20 Décembre 2020

TELECOM Nancy

Table des matières

1 Explication du code

1.1 Organisation du code

Le projet est composé de trois fichiers de codes `testaro.c`, `testaroSub.c` et `testaroFonction`.

Le fichier `testaro.c` contient la fonction principale qui lance un minuteur pour respecter le `timeout` et appelle les fonction de `testaroSub.c` pour obtenir, lire et exécuter les commandes des différents fichiers de commandes.

Le fichier `testaroSub.c` contient la fonction de test d'existence d'un fichier, de lecture ligne par ligne et de gestion de la ligne en fonction du premier caractère. Cette dernière fonction appelle les fonctions d'exécutions de `testaroFonction` selon le premier caractère d'une ligne.

Le fichier `testaroFonction.c` contient les fonctions d'exécutions avec ou sans entrée standard et avec ou sans sortie standard à comparer.

1.2 Timeout

Le Timeout sur un processus se fait à l'aide d'un signal `SIGVTALRM`. Premièrement, (lignes 30 à 32), on initialise le "signal timeout" et on lui demande d'exécuter la fonction `timer\_handler` qui donne un message d'erreur et renvoie le code d'erreur. Après, on configure le délai d'attente pour envoyer le signal `SIGVTALRM` au processus. Vu que l'on doit envoyer le signal sur l'exécution d'une seule commande, on crée un processus fils pour l'exécution de chaque commande à l'aide de la fonction `fork` (ligne 44). À chaque début d'exécution d'une commande, on met en route le timer préalablement configuré, à l'aide de la commande `setitimer`. Ainsi, si le délai d'attente de fin d'exécution est dépassé, on envoie au processus fils, le signal `SIGVTALRM` avec le message et le code d'erreur associé.

1.3 Boucle sur les fichier en arguments, test d'existence d'un fichier

Comme dit précédemment, le fichier `testaro.c` est le fichier principal de ce programme. De manière générale, le programme prend en paramètres n fichiers textes et les traites grâce à la boucle for de la ligne 43. Dans le cas où les sous-programmes renvoient un code différent de 0, alors, à l'aide de la ligne 64 `exit(WEXITSTATUS(status)`, on quitte le programme avec le code d'erreur donné par les sous-programmes.

Dans un premier temps, on doit regarder si le fichier existe : à l'aide de la fonction `testFile`, on regarde à l'aide de l'ouverture du fichier, si la fonction d'ouverture du fichier nous renvoie un fichier `null` ou non. Si le fichier est à `null`, on renvoie le code 1, sinon on continue l'exécution de `testaro` car la lecture est possible. Dans un second temps, on démarre le `timer` initialisé au préalable (entre les lignes 30 et 39).
Enfin, on analyse le code du fichier visé grâce à la fonction `analyzer` prenant en argument le nom du fichier. Elle renvoie un code qui est traité à la fin du programme.

1.4 Boucle sur les lignes à traiter et gestion du premier caractère

La fonction `analyzer`, prenant en entrée le nom du fichier, va nous permettre d'interpréter les lignes de commandes de ce fichier.

Afin d'analyser chaque ligne du fichier, il faut tout d'abord ouvrir le fichier. La séparation de chaque ligne se fait grâce à la commande `fgets` et à la boucle `while`. Tant qu'il existe une ligne à analyser, `fgets` renvoie à la variable `line` la ligne concernée (de type chaîne de caractère), s'il n'y a plus de ligne à analyser, `fgets` renvoie alors un `null` et ainsi la boucle se termine.
À la fin de l'analyse de chaque ligne, on demande à la fonction `execLine`, permettant d'exécuter une ligne, le code qu'elle renvoie. Si son code d'erreur est différent de 0, alors l'exécution du programme s'arrête et renvoie le code d'erreur à la fonction principale. Sinon, le programme continue (ligne 197).

1.5 Reconnaissance des lignes blanches

Les lignes blanches sont reconnus par le fait que la ligne possède '#' en premier caractère. Ainsi, il suffit de rejeter l'analyse des lignes commençant par ce caractère à l'aide d'une condition : si la ligne ne possède pas le caractère en début de celle-ci, alors on analyse ligne, sinon, on passe à la prochaine ligne (cf. ligne 160).

1.6 Concaténation des lignes

Pour savoir si l'on doit concaténer deux lignes entre elles, il faut savoir s'il y a le caractère '\' en fin de ligne. Tout d'abord, il faut connaître le nombre de caractères de la ligne jusqu'au caractère de retour à ligne (`\n`). On sauvegarde le nombre de caractères dans la variable `increm`. Ensuite, le programme regarde si l'avant-dernier caractère est '\'. Si oui, on enregistre la ligne dans `lastLine` et on incrémente la variable `lineJump`. Il y a trois précisions à apporter :

— On effectue un réallocation de mémoire de `lastLine` car on agrandit la ligne que l'on veut exécuter. La réallocation est donc nécessaire afin d'éviter une fuite de mémoire. Elle est d'autant plus importante s'il on concatène plusieurs centaines de lignes à la fois.

— La concaténation entre deux lignes (ou deux chaînes de caractères) se fait à l'aide de la commande `strcat` prenant en paramètre les 2 lignes et le nombre de caractères que l'on veut concaténer. Le troisième paramètre nous permet d'enlever les caractères '\' et '`\n`' de la concaténation.

— La variable `lineJump` permet de savoir au programme que la prochaine ligne (différente d'une ligne blanche) devra être concaténée avec la dernière (ligne 172).

S'il n'y a pas de '\' en fin de ligne, la ligne doit être exécutée. Si la ligne précédente possèdait le caractère '\' en fin de ligne, alors il faut effectuer une concaténation comme expliquer dans le paragraphe précédent, ensuite on exécute la ligne `lastLine` à l'aide de `execLine`. S'il n'y a pas de concaténation à faire, on exécute directement la ligne.

1.7 Traitement d'une ligne

Lorsqu'une ligne est entièrement récupérée, elle est d'abord traitée par la fonction `execLine`. Cette fonction sert à attribuer les bonnes actions à réaliser en fonction de son premier caractère :

— pour un "<" : si aucune ligne ne commençant par le même caractère n'a été rencontrée, connu par le `marqueurBufferStdin`, on réserve un espace mémoire pour un buffer qui servira en entrée standard lors de la prochaine commande rencontrée. Si une ligne à déjà été rencontrée, on récupère un espace plus grand dans lequel on enregistre la concaténation du buffer déjà connu et de la nouvelle ligne. Aucune exécution n'a eu lieu, on renvoie donc 0 pour signaler que la ligne a été traitée.

— pour un ">" : processus similaire à celui pour "<", à l'exception que le buffer contiendra l'élément de comparaison pour la sortie standard de la commande au lieu de l'entrée standard à lui passer.

— pour un "$" : on teste l'existence d'une entrée standard à passer à la commande ou d'une sortie à laquelle il faudra comparer la sortie standard de la commande, connue par les entiers `marqueurBufferStdin` et `marqueurBufferStdout`. En fonction de cela, on connaît 4 traitements avec les fonctions de `testaroFonction` qui sont décrits dans la partie suivante.

— pour tout autre caractère : la ligne n'est pas reconnue, on renvoie donc 1 pour signaler une erreur dans le fichier.

1.8 Traitement pour le caractère "$"

Comme énoncé précédemment, lorsque une ligne commence par un "$" on effectue un des 4 traitements suivant .

— *sans entrée ou sortie standard :* on appelle la fonction `execution` qui renvoie 0 si l'exécution de la commande s'est bien passée ou le code d'erreur sinon. On renvoie ce code pour dénoter la bonne exécution ou non de la commande.

— *avec entrée mais sans sortie standard :* on appelle la fonction `executionToStdin` qui renvoie 0 si l'exécution de la commande avec son entrée standard s'est bien passée ou le code d'erreur associé sinon. On pense à libérer l'espace mémoire du buffer qui vient d'être utilisé et de remettre à zéro le marqueur d'entrée standard. On renvoie le code d'exécution pour dénoter la bonne exécution ou non de la commande.

— *avec sortie mais sans entrée standard :* on appelle la fonction `execCompa` qui renvoie 0 si l'exécution s'est bien passé et si la sortie standard correspondait à celle connue par le buffer, 2 si l'exécution s'est bien passée mais la sortie standard et le buffer sont différents, ou le code d'erreur sinon. On pense à libérer l'espace mémoire du buffer qui vient d'être utilisé et de remettre à zéro le marqueur de sortie standard. On renvoie le code d'exécution pour dénoter la bonne exécution et comparaison ou non de la commande.

— *avec entrée et sortie standard :* on appelle la fonction `execCompaStdin` qui renvoie 0 si l'exécution avec l'entrée standard s'est bien passé et si la sortie standard correspondait à celle connue par le buffer, 2 si l'exécution avec entrée standard s'est bien passée mais la sortie standard et le buffer sont différents, ou le code d'erreur sinon. On pense à libérer l'espace mémoire des deux buffers qui vient d'être utilisés et de remettre à zéro les marqueurs d'entrée et sortie standard. On renvoie le code d'exécution pour dénoter la bonne exécution et comparaison ou non de la commande.

1.9 Exécution normale

Pour une exécution normale, avec `execution`, on supprime le caractère "$", on crée un processus grâce à `fork`. Dans le processus fils, on exécute la commande grâce à `execl`. Si l'exécution renvoie -1, on sait qu'il y a eu une erreur et on renvoie directement 1, le code d'erreur associé. Sinon, on enregistre le statut de sortie et on sort du processus. Le processus père attend et teste le statut de sortie. Si le statut dénote une erreur, le père le renvoie 4, le code d'erreur associé.

1.10 Exécution avec une entrée standard

Pour une exécution avec une entrée standard, avec `executionToStdin` ou `execCompaStdin`, on supprime le caractère "$", on s'occupe de l'entrée standard puis on exécute la commande attendue.
Pour l'entrée standard, on remplace le caractère initial par la chaîne de caractères "echo". On crée un fils à l'aide d'un `fork` dans lequel on change la sortie standard pour qu'il écrive dans un tube grâce à `dup`. Grâce au "echo" ajouté, on y écrit directement ce qui était stocké dans le buffer. Une fois cette exécution terminée, on ferme le processus et on teste dans le père que son exécution s'est bien passée.
Ensuite, on exécute la commande prévue : on crée un fils dans lequel on change l'entrée standard par la sortie du tube pour qu'il prenne bien l'entrée standard demandée. Une fois l'exécution terminée, on ferme le processus et on teste dans le père que l'exécution s'est bien passée.
Enfin, si tout s'est bien passé, on renvoie 0.

1.11 Exécution avec une sortie standard

Pour une exécution avec une sorteie standard, avec `execCompa` ou `execCompaStdin`, on supprime le caractère "$", on on exécute la commande attendue en la modifiant légérement en amont, puis on compare sa sortie standard avec le buffer.
Tout d'abord, on modifie la commande initiale pour qu'elle écrive sa sortie standard dans un fichier `temp.txt`. On l'exécute ensuite dans un nouveau processus. Une fois cette exécution terminée, on ferme le processus et on teste dans le père que son exécution s'est bien passée.
Pour comparer la sortie standard de la commande avec le contenu du buffer, on ouvre le fichier `temp.txt` et on récupère son contenu dans un espace mémoire alloué. On compare les deux chaînes de caractères grâce à `strcmp`.
Avant de faire un retour en fonction du résultat de comparaison, on libère l'espace alloué à la sortie reçue, puis on renvoie 0 si les deux chaînes étaient les mêmes, 2 sinon. Ce retour est prioritaire pour la fonction `execCompaStdin`.

2 Problèmes rencontrés

2.1 Compréhension du sujet

Pour les deux membres du groupes, des aspects du sujets n'étaient pas clairs, notamment dans l'utilisation des lignes débutant par des chevrons. La question majoritaire était de savoir comment l'apparition de plusieurs chevrons allait influer le fonctionnement du programme.

Nous avons décider de considérer que lorsque l'on trouvait plusieurs fois le même type de chevrons avant une exécution, les lignes concernées seraient concaténées pour ne former qu'une unique chaîne de caractère.

2.2 Communication

Dû à la pandémie de Covid-19, la communication s'est passée uniquement par Discord. Ceci a été source de problème de compréhension entre nous et de choix dans les méthodes d'implémentation.

3 Bilan horaire

Étapes	Marie-Astrid Chanteloup	Niels Tilch
Conception	5	5
Implémentation	10	12
Tests	1	2
Rédaction de rapport	7	4
Total	23	23