Projet `testsuite`

RS : Réseaux et Systèmes

Deuxième année

L'objectif de ce projet est d'écrire un programme permettant d'automatiser la procédure de test d'autres programmes. On sera amené à utiliser la plupart des appels systèmes vus en cours comme `fork(2)` et `execlp(3)` (pour lancer des commandes), `pipe(2)` et `dup2(2)` (pour modifier stdin et stdout des commandes lancées), `read(2)` et `write(2)` (pour communiquer avec les commandes lancées), `sigaction` (pour manipuler les signaux). L'appel système `chdir(2)` sera également utile pour implémenter la commande `cd`. Il est également possible que l'appel système `fcntl(2)` s'avère utile.

Informations générales

Évaluation de ce projet

- **10 points** seront attribués de manière semi-automatique par évaluation des fonctionnalités de votre programme. Après l'avoir compilé, nous utiliserons votre programme pour exécuter différentes suites de test. Vous perdrez des points si votre programme ne se comporte pas comme attendu. Pour le bon fonctionnement de ce processus (et l'attribution des points correspondants), votre programme doit respecter les codes de retour détaillés plus loin. La note *maximale* d'un projet **ne compilant pas en l'état** sur neptune est 8.
- **4 points** seront attribués en fonction de la qualité (subjective) du source de votre programme.
- **4 points** récompenseront la qualité de votre mini-rapport. Vous y détaillerez les difficultés auxquelles vous avez été confronté, et comment vous les avez résolues. Vous indiquerez également le nombre d'heures passées sur les différentes étapes de ce projet (conception, codage, tests, rédaction du rapport).
- Des points de bonifications seront attribués pour chacune des extensions implémentées (voir plus loin).

Comment rendre votre projet

Vous devez rendre un mini-rapport de projet (5 pages maximum, format pdf), votre source ainsi qu'un makefile permettant de compiler votre projet (sous le nom `testsuite`). Si vous écrivez de nouvelles suites pour tester votre programme, rendez-les également avec votre projet (avec l'extension `.suite`) : une bonification est prévue pour les projets dotés d'un ensemble de tests bien fourni.

Pour rendre votre projet, vous devez placer les fichiers nécessaires dans un répertoire sur neptune, vous placer dedans, et invoquer la commande `/home/EqPedag/quinson/bin/rendre_projet RS`

Avant le lundi 27 novembre à minuit

(la commande ne fonctionnera plus après)

Rappel

La tricherie sera **sévèrement punie**. Voir `http://www.loria.fr/~quinson/teaching.html#triche`

Informations complémentaires

Vous trouverez des informations complémentaires sur le projet et quelques exemples de fichier de suite à l'adresse `http://www.loria.fr/~quinson/teach-RS.html`.

1 Description du projet testsuite

Vous devez écrire un programme testsuite dont l'objectif est d'aider à automatiser la procédure de tests des fonctionnalités d'autres programmes. Ses entrées sont des fichiers de descriptions de scénarios de tests contenant à la fois les actions à réaliser, et leur résultat attendu.

Votre programme testsuite doit accepter un seul argument sur la ligne de commande. Il s'agit du nom du fichier décrivant les tests à mener. Si l'argument désigne un fichier ne pouvant être lu, votre programme doit s'arrêter avec un message et un code d'erreur appropriés.

1.1 Syntaxe du fichier de description

Les lignes d'un fichier de description peut être de différents types, en fonction de leur premier caractère. Le second caractère doit être un espace (ignoré par testsuite). Ces lignes peuvent être séparées par des lignes blanches, qui seront également ignorées. Un caractère \ placé à la fin d'une ligne signifie que la ligne suivante doit être considérée comme la suite de la ligne courante (voir l'exemple plus loin).

Caractère	*Signification*
'#'	Commentaire à ignorer
'$'	Commande shell à exécuter
'<'	Chaîne à copier sur l'entrée standard de la prochaine commande
'>'	Chaîne que la commande suivante doit afficher

```
1 # Ceci est un exemple de suite
2 < TOTO \
3 TUTU
4 $ cat > fich
5
6 > TOTO TUTU
7 $ cat fich
```

Dans l'exemple de gauche, la première ligne est un commentaire dont votre programme ne doit pas tenir compte.

La ligne 2 est terminée par un \. Cela signifie que la ligne 3 est en fait la suite de la ligne 2. On aurait obtenu le même effet que les lignes 2 et 3 en écrivant `< TOTO TUTU` sur la ligne 2.

Les lignes 2 à 4 signifient qu'il faut lancer la commande `cat > fich` en écrivant sur l'entrée standard du shell la chaîne de caractère "TOTO TUTU". Utilisé sans argument, le programme cat copie son entrée standard sur sa sortie standard. Ces lignes permettent donc d'écrire "TOTO TUTU" dans le fichier fich.

La ligne 5 doit être ignorée puisqu'elle est blanche. Les lignes 6 et 7 indiquent quant à elles que lorsque l'on exécute la commande indiquée ligne 7, elle doit afficher le texte donné ligne 6.

1.2 Actions réalisées par testsuite

testsuite analyse chaque ligne du fichier de description du test, et réalise les actions indiquées sur les lignes '$' en leur passant les lignes '<' sur leur entrée standard. En cas de problème avec ce fichier, testsuite termine avec le code 1.

Si l'une des commandes n'affiche pas le texte attendu par les lignes '>', testsuite l'indique par un message comprenant les deux chaînes (messages attendu et obtenu), puis termine avec le code 2.

Si l'une des commandes reçoit un signal, testsuite termine avec le code <numéro du signal>+3 (donc avec le code 14 en cas de SIGSEV puisque SIGSEV=11 sous linux).

Si l'une de ces commandes renvoie un code d'erreur différent de 0, testsuite s'interrompt en indiquant une erreur : un texte approprié est affiché, puis le testsuite termine avec le code renvoyé par la commande, plus 40 (ie, si la commande renvoie 3, il faut que testsuite renvoie 43).

Votre programme devra également être capable de détecter si le programme testé est entré en boucle infinie, et l'interrompre au besoin. Pour cela, vous armerez une alarme à 5 secondes avant le lancement de chaque commande. Si la commande ne se termine pas avant le déclenchement de l'alarme, testsuite doit produire un message d'erreur conséquent et terminer avec le code 3.

Si tout se passe comme prévu dans le fichier de test, testsuite doit retourner le code 0.

Code à renvoyer	Signification
0	La suite de test s'est bien passé
1	Problème à la lecture de la suite (fichier inexistant, ou erreur de syntaxe)
2	Une commande n'a pas renvoyé le texte attendu
3	Une commande n'a pas terminé avant l'expitation du délai de garde
4	Problème lors d'un appel système
5-40	Une commande a reçu un signal $n-4$
40-	Une commande a renvoyé le code $n-40$

1.3 Commande spéciale : cd

La commande "cd" ne doit pas être traitée de la même façon que les autres : il faut changer le répertoire de travail de testsuite et non celui d'un processus fils. Utilisez pour cela l'appel chdir(2).

2 Réalisation

2.1 Outils à utiliser

Le cœur du projet est d'exécuter des commandes dans des shells placés dans des processus fils et de communiquer avec eux. Il faut donc ouvrir deux tubes avant le fork et l'exec : l'un pour pouvoir écrire sur l'entrée standard du fils et l'autre pour lire ses sorties standard et d'erreur (que l'on fusionnera).

Pour exécuter les commandes, lancez (avec `execlp`) le programme `sh -c "ligne de shell"`.

2.2 Stratégie possible

Pour votre gouverne, voici comment j'ai procédé lorsque j'ai implémenté le projet. Vous êtes libre de suivre ce cheminement ou non.

1. Écrire le code pour trouver le nom du fichier de description. Écrire le makefile de votre projet et tester l'ensemble.
2. Écrire la boucle principale de lecture du fichier de description. Elle utilise `getline()` et appelle pour chaque ligne une fonction de traitement séparée (qui affiche son argument pour l'instant).
3. Écrire le code reconnaissant les lignes blanches.
4. Écrire le code gérant le caractère \ en fin de ligne. Cela utilise un tampon pour stocker la ligne en cours d'analyse et ne passe son tampon à la fonction de traitement que si le dernier caractère est différent de \. Sinon, elle concatène la ligne courante au tampon en supprimant le \ et le \n.
5. Écrire dans la fonction de traitement le code traitant différemment les lignes reçus en fonction de leur premier caractère (pour l'instant, simplement faire un `printf` approprié). Produire un message d'erreur pour les lignes de type inconnu et ignorer les commentaires.
6. Mettre en place un accumulateur des lignes d'entrée ('<') dans la fonction de traitement. Les lignes à passer à la prochaine commande y sont accumulées (en préservant les caractères '\n' de séparation). Faites de même pour les lignes de sortie. Factorisez le code de gestion des accumulateurs (cf. §4).
7. Il faut maintenant écrire la fonction réalisant un test. Elle doit ouvrir deux tubes (`pipe(2)`) pour récupérer stdin et stdout du fils, forker, mettre en place les tubes dans le père grâce à `dup2` puis exécuter la commande par un `execlp` lançant le binaire `sh` avec `-c` comme premier argument et la commande à exécuter comme deuxième argument. Nous allons procéder graduellement.
8. Mettez en place les tubes avec pipe, et testez votre solution. Le père envoie "toto" sur l'entrée du fils et vérifie qu'il peut lire la même chaîne de la sortie du fils. Le fils lit une chaîne sur son entrée et la recopie sur sa sortie. Quand ce test fonctionne, vos tubes sont bien en place.
9. Mettez en place l'`exec`. Commencez par exécuter la commande "cat", pour vérifier que le test précédent fonctionne encore. Changez l'exec pour exécuter la commande lue dans le fichier.
10. Dans le père copiez le contenu de l'accumulateur d'entrées sur le stdin du fils (attention aux écritures tronquées). Fermez ensuite le fichier stdio du fils (depuis le père) pour que le processus fils sache qu'il n'y a plus rien à lire.
11. Dans le père, remplissez un accumulateur de sortie de ce que le fils écrit sur sa sortie. On comparera son contenu à ce qu'on attendait plus tard, pour l'instant il suffit d'en afficher le contenu. Cette tâche est compliquée par le fait qu'on ne sait pas à priori combien de caractères il convient de lire. La condition d'arrêt est que `read` retourne 0 quand le tube a été fermé par l'écrivain. Modifiez votre père pour qu'il lise toutes les sorties du fils.
12. Placez un `wait(NULL)` après la fin de la lecture dans le père pour ramasser les processus zombies. Testez votre travail ; le plus dur est fait.
13. Mettez en place le code dans le père (à la place de `wait(NULL)`) pour récupérer le code de sortie de la commande, et terminez `testsuite` avec le bon code d'erreur si celui-ci ne vaut pas 0 ou si le fils a reçu un signal.
14. Mettez en place le code comparant la sortie obtenue à la sortie attendue. Ignorez les erreurs consistant en des différences de \n à la toute fin des chaînes.
15. Implémentez la commande `cd`.
16. Implémentez un mécanisme de délai de garde pour éviter de geler `testsuite` si le programme testé entre dans une boucle infinie.
17. Apportez les touches finales au code, nettoyez le et testez le convenablement.

3 Aller plus loin

Cette section contient quelques idées pour les curieux souhaitant aller plus loin. Vous êtes libre d'implémenter ou non les idées présentées ici, sachant que des points de bonifications sont attachés à chacune d'entre elles.

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.
3. D'autres commandes (comme `su`) nécessitent le même traitement de faveur que `cd`. Trouvez lesquelles et implémentez les.
4. Permettre aux lignes '>' d'être placées après la commande à exécuter dans le fichier de description. Cela modifie la sémantique des fichiers de suite car les lignes '>' doit alors être rattachées à la ligne '$' précédente et non à la suivante.
   Il ne faut donc activer ce comportement qu'après une ligne type `! set output post`
   (`! set output pre` permet de revenir au comportement par défaut).
   Ensuite, on ne peut plus vérifier que la commande a bien affiché ce qu'elle devait juste après l'avoir exécutée puisque qu'on a pas encore lu les lignes '>' lui correspondant. Cette comparaison doit être faite lorsque l'on est sûr de ne plus pouvoir trouver de ligne de sortie dans le fichier : lorsque l'on rencontre la commande suivante, ou à la fin du fichier.
5. 5 secondes de délai de garde peut être un mauvais choix dans certains cas. Le mieux est d'ajouter des lignes telles que `! 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.
7. Certaines commandes doivent déclencher un signal ou retourner une valeur différente de 0. Pour cela, on utilise des lignes `! expect signal <numéro du signal attendu>` ou
   `! expect return <valeur attendue>`. Cela ne s'applique qu'à la commande suivante.
8. Implémenter la commande `!include <fichier>`.
9. Il n'est pas toujours utile de passer par un shell pour lancer la commande. S'il s'agit d'un simple programme, on peut économiser le shell en l'exécutant directement. Il s'agit des commandes ne comprenant pas un seul des caractères suivants : | <> " ' ` $
10. Les commandes `export nom=valeur` et `unset nom` modifient l'environnement de `testsuite` (qui est hérité par les commandes lancées).
11. Le mécanisme décrit ici ne convient pas pour tester des programmes interactifs. Pour ceux-là, il faut pouvoir leur donner plusieurs séries d'entrées entrecoupées de valeur à lire absolument. On doit pour cela mélanger les lignes '<' et les lignes '>' dans le fichier de description et vérifier que l'ordre est respecté lors de l'exécution du fils.
12. Pour lancer certaines commandes en tâche de fond, on peut utiliser des lignes '&'. La syntaxe est la même que pour les lignes '$', sauf que le travail fait habituellement dans le père est fait dans un fils (ou dans un thread, pour les plus courageux d'entre vous). La commande est donc lancée dans le fils du fils de contrôle, et le fils de contrôle doit communiquer avec le père pour l'informer du résultat du test. Une autre solution est d'utiliser `select(2)` pour communiquer avec plusieurs fils.
13. Parfois, on ne peut pas prédire la sortie du programme à l'avance (par exemple si le pid du processus ou la date en font partie). On souhaite pouvoir utiliser des expressions régulières. On introduit les lignes '~' qui ressemblent aux lignes '>'. Il faut alors passer la sortie obtenue de la commande à une commande `perl -e 'm/<argument de la ligne ~>/'`. Elle renvoie vrai si la sortie respecte l'expression régulière (et est donc valide).
14. Si vous avez d'autres idées d'extensions, vous êtes les bienvenus pour les implémenter (et les documenter dans votre rapport).