Projet testaro – système de tests automatiques

RS : Réseaux et Systèmes

Deuxième année

L'objectif de ce projet est d'écrire un programme qui permette d'automatiser les test d'autres programmes. Vous utiliserez les appels systèmes vus en cours : `fork(2)`, `execlp(3)` (ou `execvp(3)`), `pipe(2)`, `dup2(2)`, `read(2)`, `write(2)`. L'appel système `chdir(2)` pourrait également être utile pour implémenter le changement de répertoire.

1 Logistique

Groupes. Apprendre à travailler en groupe fait partie des objectifs de ce projet. Il vous est donc demandé de travailler en **binôme** (dont la composition est libre – il est possible de mélanger deux groupes de TD, et étudiants de TELECOM Nancy et étudiants de l'école des Mines). Il est **interdit** de travailler seul (sauf pour au plus un étudiant, si vous êtes un nombre impair). Un ingénieur travaille rarement seul.

Langage. Le projet devra être codé en langage C et devra compiler sans intervention extérieure à l'aide de `make` (et d'un `Makefile` correspondant).

Tests automatiques. Une part importante de l'évaluation du projet sera faite à l'aide d'une batterie de tests exécutés sur une distribution GNU/Linux (Debian Buster précisément). Il est donc très important, au cours de votre travail, d'accorder une large part à vos propres tests et de vérifier leur bonne exécution dans le même environnement que celui d'évaluation. Plusieurs «tests blancs» seront également organisés au cours du projet : votre code sera récupéré, compilé, et testé sur des cas de tests qui feront partie de ceux utilisés pour l'évaluation finale, et les résultats de ces tests vous seront communiqués. **Il est donc indispensable de commencer à travailler tôt pour bénéficier de ces «tests blancs» et avoir la garantie que votre projet fonctionne correctement.**

Rapport. Vous devez rendre un mini-rapport de projet (5 pages **maximum** hors annexes et page de garde, format **pdf**). Vous y détaillerez vos choix de conception, les extensions que vous aurez développées, 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, implémentation, tests, rédaction du rapport) par membre du groupe.

Évaluation

- **10 points** seront attribués de manière automatique par les tests. Après compilation, nous utiliserons votre programme pour exécuter différentes suites de tests. Vous perdrez des points si votre programme ne se comporte pas comme prévu. Votre programme doit donc bien respecter les codes de retour qui sont détaillés plus loin dans ce document.
- **3 points** seront attribués en fonction de la qualité subjective du code source de votre programme
- **2 points** seront attribués en fonction de la qualité du mini-rapport.
- Des points supplémentaires seront attribués pour chacune des extensions implémentées.

Soutenances. Des soutenances pourront être organisées (éventuellement seulement pour certains groupes, par exemple s'il y a des doutes sur l'originalité du travail rendu). Vous devrez nous faire une démonstration de votre projet et être prêts à répondre à toutes les questions techniques sur l'implémentation de l'application.

Plagiat et aide extérieure au binôme. Si, pour réaliser le projet, vous utilisez des ressources externes, votre rapport doit les lister (en expliquant brièvement les informations que vous y avez obtenus). Un détecteur de plagiat [1] sera utilisé pour tester l'originalité de votre travail (en le comparant notamment aux projets rendus par les autres groupes). Toute triche sera sévèrement punie.

Concrètement, il n'est pas interdit de discuter du projet avec d'autres groupes, y compris de détails techniques. Mais il est interdit de partager, ou copier du code, ou encore de lire le code de quelqu'un d'autre pour s'en inspirer.

1. `http://theory.stanford.edu/~aiken/moss/`

Informations complémentaires. Des informations complémentaires (foire aux questions par exemple) pourront être fournies sur le dépôt git projet-2021-testaro du groupe RS2021 du gitlab de l'école : `https://gitlab.telecomnancy.univ-lorraine.fr/rs2021/projet-2021-testaro`. Ces informations complémentaires doivent être considérées comme faisant partie du sujet. **Il est donc conseillé de surveiller cette page régulièrement**.

Questions. Vos questions éventuelles peuvent être adressées à `jean-philippe.eisenbarth@loria.fr`. Les réponses (et les questions correspondantes) pourront être publiées dans la FAQ sur la page du projet.

2 Description du projet testaro

Testaro est une expression en esperanto qui signifie *ensemble de tests* en français `;-)`

Vous devez écrire un programme `testaro` 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 `testaro` 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.

2.1 Syntaxe du fichier de description

Chaque ligne du fichier de description peut être l'une des suivantes :

- Une ligne blanche (vide/saut de ligne ou ne contenant que des caractères blancs tels que des espaces, tabulations) qui doit être ignorée.
- `# du texte` un commentaire à ignorer.
- `$ commande` Une commande shell à exécuter.
- `< chaine de caractères` Une chaîne de caractère à copier sur l'entrée standard de la prochaine commande.
- `> chaîne de caractères` Une chaîne de caractère que la commande suivante doit afficher.

2.2 Exemple de fichier de description

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

Dans l'exemple ci-dessus, 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 la suite de la ligne 2.

Les lignes 2 à 4 signifient qu'il faut lancer la commande `cat > fich` en écrivant sur l'entrée standard de la commande 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.

2.3 Déroulement de l'exécution de `Testaro`

`Testaro` 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, `testaro` termine avec le code 1.

Si l'une des commandes n'affiche pas le texte attendu par les lignes '>', `testaro` 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, `testaro` 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, `testaro` s'interrompt en indiquant une erreur : un texte approprié est affiché, puis le `testaro` termine avec le code renvoyé par la commande, plus 40 (ie, si la commande renvoie 3, il faut que `testaro` 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, `testaro` doit produire un message d'erreur conséquent et terminer avec le code 3.

Et pour finir si une erreur avec un appel système intervient (une erreur dans l'appel à `fork()` par exemple), votre programme devra renvoyer le code 4.

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

Code à renvoyer	Signification
0	La suite de tests s'est bien passée
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'expiration 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 $n-40$

2.4 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 `testaro` et non celui d'un processus fils. Utilisez pour cela l'appel système `chdir(2)`.

3 Travail à réaliser

3.1 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 une fonction de la famille `exec`) le programme `sh -c "ligne de shell"`.

3.2 Stratégie possible

Voici quelques conseils pour avancer dans le projet si vous êtes bloqués :

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 peut afficher 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 sur 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 stdin 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 `testaro` 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 `testaro` si le programme testé entre dans une boucle infinie.
17. Apportez les touches finales au code, nettoyez le et testez le convenablement.

4 Extensions possibles

Les extensions réalisées seront valorisées (mais il faut d'abord réaliser correctement tous les points décrits ci-dessus). Elles doivent être décrites dans le rapport. Comme indiqué plus haut la partie obligatoire du projet sera notée sur 16, pour obtenir une meilleure note il faudra donc implémenter plusieurs des extensions présentées ci-dessous.

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 `testaro` (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).

5 « Rendu » du projet.

Le « rendu » du projet se fera via l'instance Gitlab de TELECOM Nancy. En dehors de la configuration correcte de votre dépôt, il n'y a rien à faire le jour de la fin du projet. Une page web (dont l'adresse sera communiqué sur le site du cours), et les tests blancs, vous permettront de vérifier la bonne configuration de votre dépôt.

Pour créer votre projet, il faut aller sur `https://gitlab.telecomnancy.univ-lorraine.fr/projects/new`. Votre *Project name* doit commencer par `rs2021-` et contenir les noms des deux étudiants du binôme (par exemple `rs2021-dupont-durand`). Son *Visibility Level* doit être *Private*.
Une fois le projet créé, allez dans Settings, Members, et ajoutez Jean-Philippe Eisenbarth (@Jean-Philippe.Eisenbarth.2) avec *Developer* comme *role permission*. Ajoutez également votre binôme.

Votre dépôt Git doit contenir, à la racine du dépôt :

— Le code source de votre projet (éventuellement dans un dossier `src`)

— Un ficher `AUTHORS` listant les noms et prénoms des membres du groupe (une personne par ligne)

— Un `Makefile` (**à la racine !**) compilant votre projet en créant un fichier exécutable (**à la racine**) nommé `testaro`

— Un fichier `rapport.pdf` contenant votre rapport au format PDF.

6 Calendrier

Des «tests blancs» seront effectués à au moins trois reprises, le **12/11/20**, le **26/11/20**, et le **14/12/20**. Votre code sera récupéré, compilé, et testé sur des cas de tests qui feront partie de ceux utilisés pour l'évaluation finale. Les résultats vous seront communiqués, mais ne seront pas pris en compte pour l'évaluation finale. L'expérience montre que l'environnement de développement et d'exécution (architecture, système, version du compilateur, ...) peut influencer les résultats et mettre en évidence des bugs qui pourraient ne pas être visibles sur vos machines. Il est donc très utile de commencer à travailler tôt

pour bénéficier de ces «tests blancs». Aucune réclamation ne pourra être acceptée si votre programme ne se comporte pas correctement dans l'environnement d'évaluation, puisque vous aurez pu bénéficier de plusieurs «tests blancs» avant le rendu du projet.

La version finale de votre projet est à rendre pour le **dimanche 20/12/2020 à 23h59**. Il sera récupéré directement sur vos dépôts git. Vous n'avez donc pas d'action particulière à effectuer pour *rendre* le projet, mais vous devez vous assurer que les fichiers requis sont bien présents. Une bonne manière de vérifier que tous les fichiers sont bien présents sur le dépôt est de réaliser un nouveau *checkout* et d'en vérifier le contenu. Les groupes dont le projet ne pourra pas être récupéré correctement seront évidemment sanctionnés.