UNIVERSITÉ DE LORRAINE

20 DÉCEMBRE 2020

Projet Testaro
Système de test automatique

Auteurs :

Mohammed KRIMI
Benjamin PEZET

1 Introduction

Ce projet est proposé dans le cadre du module *Système* à Telecom Nancy. En travaillant en binôme, pendant une durée de 2 mois, nous devions mettre en oeuvre les acquis du module afin de répondre à la problématique du projet.
Le but de l'OS est de diriger l'utilisation des ressources d'un ordinateur par un système applicatif. Le but du projet testaro est similaire, il demande d'automatiser une suite de tests de fonctionnalités d'autre programmes décris dans un fichier texte.
Contrairement au test manuel, le test automatisé s'exécute sans l'intervention de l'humain, et se base sur une suite de solutions informatiques. Il a pour objectif de simplifier au maximum les efforts des interventions humaines à travers les scripts déployés, en les exécutant l'un après l'autre, il compare les résultats du test à ceux attendus.
Le test automatique permet aussi de vérifier en permanance les objets de test, leur rapidité d'exécution, donne à l'utilisateur le pouvoir de tester son application le plus de fois possible et de la meilleure manière.
Cependnat, le test automatisé n'est pas facile à implémenter, puisqu'il nécessite une suite des étapes à suivre parfaitement pour aboutir à des résultats promettants. En effet, le gain en temps et en ressources qu'ils permettent, nécessitent un vrai investissment en les comparant aux tests manuels, en particulier leur mise en place.
Ce projet se base juste sur l'utilisation des appels système dans le langage C afin de permettre la création, la gestion, l'exécution des processus système. Mais le programme *Testaro* doit prendre un seul argument dans la ligne de commande, qui est le nom du fichier contenant les instructions des tests à effectuer (fichier de description).

2 Conception

2.1 Étapes de codage de Testaro

Avant tout, nous nous sommes mis d'accord sur la forme finale du code et sur son exécution. Nous avons par la suite implémenté les étapes proposées par le sujet une après l'autre.

2.2 Code Testaro

Le fichier *traitement.c* contient les fonctions suivantes :

— **show(char* s)** : cette fonction prend en paramètre un chaîne de caractère et l'affiche sur la sortie standard.

— **int isEmpty(char* s, int nb_char)** : cette fonction nous permet de savoir si la ligne est vide ou pas, elle retourne 0 si c'est le cas, 1 sinon.

— **int isSeperated(char* s, int nb_char)** : cette fonction nous permet de savoir si une ligne est séparée en deux (ou plusieurs) lignes afin de la concaténer après.

— **copie(char* ligne, char* entree_sortie)** : cette fonction joue le rôle d'accumulateur d'entrée/sortie. La ligne est copiée (accumulée) dans une variable passée en second paramètre.

— **traitement_ligne(char* ligne, char* entree, char* sortie)** : cette fonction nous permet d'appliquer les traitements nécessaires à chaque ligne. En effet, elle regarde le premier caractère de chaque ligne et lui applique le traitement convenable.

— **realisation_test(char* ligne, char* entree, char* sortie)** : cette fonction nous permet de réaliser le test. En effet, elle crée deux processus, mets en place deux tubes : le premier s'occupe de l'entrée du fils et le deuxième de sa sortie. Elle permet au fils de récupérer le contenu envoyé par le père sur son entrée standard et la transmettre à sa sortie. Le code sortie du fils est utilisé pour savoir si tout s'est passé comme prévu.

— **execution(char* ligne)** : cette fonction est un **execlp()** amélioré puisqu'elle ne prend qu'en paramètre la commande à exécuter (c'est juste pour nous faciliter le travail lors du codage).

— **traitement(FILE* file)** : cette fonction permet de traiter le fichier ligne par ligne. Si une ligne de vide, elle passe à la ligne suivante, si une ligne se termine par '\' elle la concatène avec la ligne suivante, tout en appelant sur chaque ligne la fonction ci-dessous :

— **traitement_ligne()** qui permet en parallèle de faire le nécessaire pour la réalisation des tests.

3 Difficultés rencontrées

3.1 Concaténation

La première version implémentée répondait parfaitement au fichier description donné dans le sujet. En créant d'autres fichiers de description, nous nous sommes rendus compte que la concaténation ne fonctionnait que pour 2 lignes séparées. Une concaténation de plusieurs lignes étant donc impossible avec cette version.

3.2 Mise en place des tubes

La notion de pipes utilisée dans ce projet est plus délicat que ce que nous avions vu jusqu'à présent. En effet, il fallait mettre en place deux tubes, le premier s'occupant juste de la liaison entre ce que remet le père dans l'entrée du fils (entrée du tube), et récupère ça sur la sortie du pipe, puis il fallait le copier dans le deuxième tube afin de permettre sa comparaison avec ce qui était remis par le père.

3.3 Le mécanisme du délai de garde

L'une des plus grandes notions de l'OS est celle des alarmes et signaux permettant la mise en oeuvre d'un système de délai. Pendant les travaux pratiques, ces notions n'étaient pas trop détaillées. Sachant qu'il existe plusieurs possibilités de les mettre en place, il nous était difficile de trouver celle correspondant à notre projet.

3.4 Le travail en groupe avec la crise sanitaire

Le contexte de la crise sanitaire est frustrant, il affecte sur tous les niveaux le travail pédagogique. En conséquent, il rend plus délicat le travail en groupe, surtout les réunions que nous étions censés en présentiels à l'école.

4 Solutions optées

4.1 Concaténation

Nous avons opté pour une solution à base récursive, elle permet de concaténer *n* lignes consécutives. Cette fonction concatène en premier lieu les deux premières lignes en enlevant le '\' qui se trouve à la fin de la première ligne. Ensuite, elle regarde si la ligne suivante se termine également par un '\', si c'est le cas, elle refait le même traitement que précédemment, sinon, elle stocke la ligne concatenée dans une variable pour les traitements complémentaires.

4.2 Les tubes

Afin de mettre en place les tubes, nous avons tenté plusieurs alternatives. Nous avons commencé par faire une analogie avec le td, mais c'était un peu délicat, vu qu'il fallait mettre en place deux tubes juste pour la gestion de l'entrée et la sortie standard du fils. Du coup, après une discussions avec d'autres groupes sur le problème de ces deux tubes, nous avons pu décrire le problème avec un schémac ce qui nous a facilité la mise en place du code.. Après la mise en place du code corrspondant, nous les avons testé à travers les tests proposés par le sujet.

4.3 Les alarmes

Après plusieurs recherches, nous avons trouvé plusieurs solutions présentés dans les forums. Ils nous ont permis de comprendre mieux la notion d'alarme. Cependant ces solutions n'était pas celles qui pouvaient nous éviter que testaro se gele. Alors, nous nous sommes redirigé vers les exemples du cours, qui ont était plus utiles dans le cadre du projet. C'est pour ça, le système de délai que nous avons mis en place est inspiré des explications et exemples du cours, et il répond parfaitement à la problématique.

4.4 Le travail en groupe

Nous avons mis en place un serveur discord, où nous avons mis plusieurs salons, ainsi que l'intégration d'un BOT gitlab envoyant des notifications directement après les commits. Pour la répartition du travail, nous avons fais de sorte que chacun fais uune étape de la stratégie proposées par le sujet. Mais vu qu'il y avait des étapes délicates, nous avons fais plusieurs séances de travil sur discord de 2h ou plus afin d'essayer de résoudre ensemble les problèmes affrotés. Les réunions nous ont aussi permis d'éviter des éventuelles effets tunnels, qui ont du affecté le rythme du travail du groupe.

5 Extensions

Les deux premières extensions proposées ont pu être réalisés.

5.1 Les lignes 'p'

Puisque les lignes sont déjà différenciées par leur premier caractère, il suffisait de rajouter un test pour détecter une ligne commençant par le caractère 'p'.

5.2 Le numéro des lignes

En initialisant une variable entière à 1, et en l'incrémentant à chaque fois que le programme lit une nouvelle ligne, on peut afficher la ligne correspondant à l'erreur. Cela est rendu posible en passant cet entier en argument dans les fonctions proposant des codes d'erreur de sortie.

6 Temps de travail

Étape	**Temps passé**
Lecture du sujet et compréhension des objetcifs	45min
1. Ouverture du fichier + Makefile	10min
2. Lecture du fichier	5min
3. Lignes blanches	20min
4. Concaténation	3h
5. Gestion premier caractère (printf)	10min
6. Accumulateur	1h
7-8. Mise en place des tubes	4h
9. Mise en place de l'exec	1h
10. Accumulateur père vers fils	3h
11. Accumulateur sortie du père	2h
12. wait(NULL)	5min
13. Code de sortie	15min
14. Comparaison sorties	15min
15. Commande cd	30min
16. Mécanisme de délai	2h
17. Touches finales	30min
1+. Lignes 'p'	15min
2+. Numéro de lignes	15min
Rédaction du rapport	2h
Total	**21h35min**