

Rapport pour le projet de Systèmes d'exploitation

STEIN Eloïse

MULLER Ludovic

9 avril 2017

Introduction

Ce document a pour but de décrire le déroulement de notre projet. Nous avons travaillé ensemble sur la réalisation de la commande **detecter**, qui lance périodiquement un programme et détecte les changements d'état.

Ce rapport contient l'ensemble des explications sur les différents choix que nous avons faits au cours de ce projet. Nous allons expliquer, le plus clairement possible, pourquoi nous avons fait ces choix et les difficultés rencontrées.

Les options

Au niveau des options, on a utilisé **getopt** et un **switch**.

Les options prises en comptes sont les suivantes :

- l'option **-i** donne l'intervalle entre deux exécutions du programme, en millisecondes (par défaut : 10 000 ms),
- l'option **-l** donne le nombre d'exécutions du programme (par défaut 0, c'est-à-dire sans limite),
- l'option **-c** affiche le code de retour lors de la première exécution et l'affiche à nouveau si celui-ci change au cours des autres exécutions,
- l'option **-t** affiche la date et l'heure au cours de chaque exécution, dans le format spécifié.

Structure Buffer

Nous avons choisi de créer une structure **Buffer** pour gérer aisément la mémoire. Au début, nous avons décidé de réallouer à chaque fois de la mémoire à notre buffer lorsque celui-ci était rempli. Mais on s'est vite rendu compte que c'était compliqué et moins efficace. Du coup, on a décidé de faire une liste chaînée de

buffer. Quand notre buffer est plein, vu que celui-ci pointe sur un nouveau buffer vide, on va pouvoir y accéder facilement et donc le remplir.

On remplit le buffer morceau par morceau (de taille `BUFFER_SIZE`, défini à 1024), jusqu'à ce qu'il n'y ait plus rien à lire. En même temps, on effectue la comparaison, pour savoir s'il y a eu ou non un changement par rapport au contenu précédent du buffer, au quel cas on affichera le contenu du buffer.

Suite à cela, on a défini plusieurs fonctions propres à cette structure (`new_buffer()`, `free_buffer()`, `print_buffer()`, ...), ces fonctions nous permettent de gérer nos buffers plus simplement et de rendre notre code bien lisible et efficace.

Le découpage du code

Nous avons fait le choix de ne pas découper le code en différents fichiers `.h` et `.c`, car nous pensons que la taille de ce programme n'est pas suffisamment élevée pour cela et que cela ne ferait que complexifier la lecture du code.

Nous avons cependant regroupés ensemble toutes les fonctions travaillant sur la structure `Buffer`, ce qui permet de les retrouver rapidement.

Gestion des erreurs

Pour gérer les erreurs, nous avons défini les macros suivantes :

- `CHECK_ERR` permet de gérer les erreurs avec les primitives systèmes, on affiche l'erreur avec `perror`.
- `CHECK_ERRVALUE` permet de gérer les erreurs pour une valeur précise, on affiche un message d'erreur sur la sortie d'erreur standard.
- `CHECK_NULL` permet de gérer les erreurs au niveau des allocations de mémoires et de localtime.

Tests et couverture du code

Pour avoir une couverture de code de 100% nous avons ajoutés quatre tests dans le fichier `test-200.sh` :

- un test qui permet de tester ce qui se passe avec une option inconnue
- un test qui vérifie ce qui se passe lorsque l'on utilise l'option `-t` avec une chaîne de caractères vide
- un test qui vérifie si une sortie qui ne change pas n'est affiché qu'une seule fois

- un test qui permet de faire échouer le `execvp` en lançant une commande inconnue.

Pour vérifier si le `execvp` a échoué, on envoie un signal `SIGUSR1`. Le parent, va vérifier si le fils a quitté avec un code de retour ou s'il a été interrompu par autre chose, comme un signal par exemple, dans quel cas il va arrêter le programme.

Concernant la couverture totale du code, nous n'avions pas trouvé d'autre moyen que d'écrire certaines parties du code dans des macros, comme les vérifications d'erreurs ou bien le code du fils avec la macro `FILS()` (il semblerait que sur turing la version de `gcov` ne soit pas à jour ou bien qu'elle n'est pas configurée correctement afin de prendre en compte le code du fils). Cependant introduire des macros peut poser certains problèmes, du fait qu'il n'y a aucune vérifications sur les types, etc. C'est la raison pour laquelle nous avons travaillé sur un petit script shell `expand_macros.sh` qu'il suffit de lancer dans le répertoire du projet, et qui va tout simplement appliquer les différentes macros sur le code, et fournir le résultat dans un fichier appelé `detecter_macroless.c`, où il sera alors possible de voir à quoi ressemble le code généré. Le script dit également si le code généré compile correctement ou non. Nous avons donc une couverture du code de 100% sur turing avec l'aide de ces macros, qui ont pu être toutes vérifiées grâce au script que nous avons écrit.

La gestion de la mémoire

Nous avons modifiés les scripts de tests afin de prendre en compte la variable `$V` lors de l'exécution des tests. Cette variable contient ou non la commande `valgrind`, passée lors de l'exécution de certaines tâches du `Makefile`. Nous le passons pas `valgrind -q` mais simplement `valgrind` car il est très rapide de chercher dans les fichiers de logs les résultats de `valgrind` et voir s'il y a eu ou non des soucis avec la mémoire, qui ne serait par exemple pas entièrement libérée. Nous avons donc ajouté directement un retour sur le terminal en modifiant le `Makefile` pour dire s'il y a eu ou non des soucis de mémoire lors de l'exécution des tests avec `valgrind`.

Répartition des tâches

Ludovic s'est occupé du `getopt` et du parsing des arguments. Eloïse s'est occupée au début de la comparaison des sorties standards en créant une première version avec la méthode de réallocation. Nous avons réfléchi ensemble à la structure `Buffer` et défini ensemble toutes les fonctions associées.

Conclusion

Ce projet nous a permis de bien comprendre le fonctionnement et la gestion des processus, la gestion de la mémoire, comment bien optimiser son code et aussi qu'il est vraiment essentiel de bien gérer les cas d'erreurs.