Projet Système

Jordane Masson - Nathan Bonnaud - Sarah Portejoie

Semestre 5

1 Introduction

Dans le cadre de l'UE programmation système, nous avons dû réaliser un projet se divisant en deux parties.

La première partie (partie A ndlr.), nous devions réaliser les fonctions basiques permettant la création d'une structure de données de tests, ainsi que les fonctions permettant d'ajouter des tests à cette dernière.

La seconde partie (partie B ndlr.) consiste en la réalisation d'une fonction exécutant chaques tests séquentiellement, et d'afficher le résultat du test sur la sortie standard, dans un fichier logfile ou vers une commande en fonction des paramètres d'exécution du programme.

Pour réaliser ce travail, nous avons utilisé l'outil collaboratif et de versionning GitHub. Il nous a également été nécessaire d'utiliser des outils tel que valgrind ou gdb pour repérer d'éventuelles fuites mémoires.

2 Partie A

Cette partie du projet a initialement débuté sur une réflexion sur le contenu de la structure de données que le framework devra manipuler.

Ensuite, nous avons implémenté la fonction s'occupant de la libération de l'espace mémoire occupé par la structure de données.

Puis nous avons implémenté les différentes fonctions permettant l'ajout de tests à la structure de données de ce framework.

2.1 Structure testfw_t du framework

Le choix des champs d'une telle structure a été induite par les arguments de la fonction testfw_init().

Nous avons ajouté d'autres champs facilitant par la suite l'implémentation des fonctions.

La structure est composée des champs suivants:

- Un tableau de pointeurs vers des fonctions de type testfw_func_t.
- Le nombre de tests contenus dans la structure du framework.
- Le nom de l'executable.
- Le nom du fichier log, passé en paramètre lors de l'initialisation de la structure
- La commande, passée en paramètre lors de l'initialisation de la structure
- Un entier définissant le temps maximum d'un test avant d'être en "timeout"
- La variable booléenne silent permettant d'avoir un affichage ou non sur le terminal.
- La variable booléenne verbose servant à afficher plus d'informations sur le fonctionnement interne du framework de test.

2.2 Fonction testfw_init()

Initialement, nous faisions les allocations mémoires de chaques champs de la structure via des mallocs, puis copions le contenu des chaînes de caractères via des strcpy(), cependant, cette technique était laborieuse et nous avons choisi de choisir une autre technique. C'est cette dernière qui sera expliquée dans le paragraphe suivant.

La première allocation mémoire est celle de la structure de donnée testfw_t. Nous testons si ce pointeur a bien été alloué comme afin par exemple d'éviter les problèmes de mémoire pleine, puis nous allouons le champs correspondant au tableau de pointeurs sur les fonctions de tests. Il s'agit donc d'une variable (struct test_t **).

Cependant, le code prenait beaucoup de place car il fallait tester le retour de strlen qui renvoie null, si chaîne null. Cela implique donc que les mallocs posaient problèmes.

Nous avons donc décider d'allouer dynamiquement la mémoire des chaînes de caractéres à l'aide des fonctions 'asprintf' pour simplifier le code. Nous appelons la fonction asprintf que dans les cas où la variable correspondante n'est pas nulle pour lui attribuer la valeur en paramètres. Sinon nous laissons cette variable à NULL.

Ensuite, nous avons géré le cas des modes d'affichage des tests.
Le mode silent étant prédominant sur le mode verbose, nous désactivons ce mode si le mode silent est passé en paramètre d'exécution du programme.

2.3 Fonction testfw_free()

Après avoir alloué la mémoire sur la création d'une variable de type (struct testfw_t), il nous faut libérer l'espace mémoire correspondant pour éviter toute fuite. Nous libérons donc les champs un par un, puis nous libérons également la variable primitivement initialisée.

2.4 Fonction testfw_length()

Grâce à notre choix d'implémentation de la structure de données du framework, il nous est suffisant de retourner la valeur entière correspondante (ie. nb_tests)

En effet, cette fonction retourne le nombre de tests enregistrés dans la variable tests de type struct test_t **). Cette dernière étant incrémentée à chaque ajout d'un test dans une autre fonction (ie. testfw_register_func()).

Nous avons choisi cette implémentation de la structure pour simplifier la récupération du nombre de test (mais une simple division entre la taille de la structure contenant les tests et celle d'un test aurait suffit)

2.5 Fonction testfw_get()

La fonction get permet de récupérer un test enregistré dans une variable de type (struct testfw_t) et retourne un pointeur sur celle-ci. A partir d'un entier, on récupère le test souhaité.

Il est important de vérifier que le numéro de test passé en paramètre ne dépasse pas le nombre de tests enregistrés, et n'est pas négatif.

2.6 Fonction testfw_register_func()

La fonction testfw_register_func permet d'enregistrer un test à partir d'un pointeur sur une fonction de type (testfw_func_t), d'une chaîne de caractères représentant le nom de cette fonction ainsi que la suite dans laquelle celle-ci est contenue.

Nous commençons par initialiser dynamiquement la variable de type (struct test_t) à l'aide d'un malloc(). Ensuite, nous allouons deux variables de type (char *) contenant le nom (resp. la suite) de ce test. Il est nécessaire de les allouer, pour éviter des dysfonctionnements si les variables passées en paramètre de cette fonction sont locales.

Ensuite, nous faisons un realloc du tableau contenant l'intégralité des tests du framework pour pouvoir ajouter celui-ci.

Si ce realloc échoue, nous faisons un free() de toutes les variables précédemment allouées et faisons un return NULL.
Sinon, le champs "tests" de la structure fw est égal à ce nouveau pointeur, on y ajoute le test et incrémentons le nombre de tests contenus dans la structure du framework.

2.7 Fonction testfw_register_symb()

La fonction testfw_register_symb() permet d'enregistrer une nouvelle fonction de test uniquement à partir d'un nom de test et de suite.
Elle retourne un pointeur sur la structure permettant d'enregistrer le test.

Tout d'abord, nous vérifions si les différentes variables passées en paramètres (ie. (struct testfw_t), (char * suite), (char * name)) ne sont pas des valeurs nulles. Si c'est le cas, nous sortons de la fonction et renvoyons NULL.

Dans le cas contraire, nous créons une nouvelle variable de type (char *), à laquelle nous lui attribuerons une chaîne de caractères suivant ce format:
$SUITE_$NAME

Grâce à cette chaîne, nous pouvons utiliser la fonction suivante :
*void *dlopen(const char *filename, int flag);*

Avec la valeur contenue dans fw->program en temps que premier paramètre, et *RTLD_NOW* comme flag.

Grâce à l'appel de cette fonction, nous pouvons récupérer la bibliothèque dynamique depuis le premier paramètre de cette fonction.

La fonction dlsym() nous permet de récupérer une fonction pouvant se trouver dans une librairie dynamique (.so) ou un exécutable particulier (type PIE), selon la correspondance des symboles passés en second paramètre.

Grâce à la valeur retournée par dlsym, nous pouvons renvoyer la valeur de retour de la fonction suivante : *testfw_register_func(fw, suite, name, (testfw_func_t) func);*

Nous ne fermons pas l'appel de la dynamic library à l'aide de la fonction dlclose(). Ne passant pas forcément "NULL" en premier paramètre de dlopen(), le descripteur renvoyé par dlopen() ne sera pas un descripteur du programme principale.

Le problème étant que nous devons être sûr que la fonction issue de dlsym() n'essaiera pas d'exécuter du code de la bibliothèque partagée après la fermeture de l'accès à cette librairie.

2.8 Fonction testfw_register_suite()

Finalement, nous avons dû nous occuper de la dernière fonction de cette première partie, testfw_register_suite(). Celle-ci prend en paramètre une variable de type (struct testfw_t) et une autre de type (char *).

Elle enregistre tout un ensemble de tests à partir de la variable (char *) passée en paramètre. Pour cela, nous utilisons une commande:

nm –defined-only (fw->program) | cut -d ' ' -f 3 | grep "(ŝuite)_ "

En effet, celle-ci nous permet de récupérer tous les tests au format 'suite_foo' avec 'suite' la variable (char *) en paramètre.

Exécuté à l'aide de popen, nous parcourons chaque ligne du fichier (FILE *) récupéré (un seul test différent étant présent par ligne) jusqu'à atteindre la fin du fichier.

Pour chaques chaînes de caractéres ainsi récupérées, nous connaissons la taille de la première partie (ie. suite, étant passé en paramètres et égal à strlen(suite)) et qu'il y a un caractère '_' pour séparer le nom de la suite et du test. Le nom du test à récupéré est donc la chaîne de caractère se trouvant à la position strlen(suite)+1 jusqu'au caractère de saut à la ligne.

Maintenant que le nom du test est isolé, et que nous connaissons donc ce dernier ainsi que le nom de la suite, nous pouvons appeler la fonction testfw_register_symb((struct testfw_t), (char *) suite, (char *) name).

3 Partie B

3.1 Fonction testfw_run_all()

Pour notre implémentation de la fonction run_all, nous avons choisi de manipuler principalement des files descriptors. Nous commençons par tester la validité des paramètres, puis regardons si certains modes spécifiques (logfile, commande) sont activés.

Ensuite, nous parcourons chaque test du framework via une boucle for.
A chaque itération de cette boucle, nous commençons par créer un fork() du processus principale, et conservons la valeur pid_t du fils dans une variable nommée "child_pid".

Dans le fils, nous fermons tous les file descriptors si le programme n'a pas été exécuté en mode verbose, ou a le mode silent. Cela nous permet de filtrer les sorties des tests.
Ensuite, nous quittons le processus avec la valeur retournée par l'exécution de la fonction.

3.1.1 Gestion du mode d'affichage 'normal'

Dans le cas normal d'affichage du framework, nous commençons par créer des copies des sorties standards et d'erreurs à l'aide de la fonction dup() qui prend le premier 'slot' de file descriptor valide. Dans le fils, comme l'option verbose n'est pas activée, nous fermons les copies des sorties standards et d'erreurs ainsi que celui du logfile.

Dans le père, nous commençons par initialiser le timer servant à calculer le temps d'exécution du fils. Nous créons aussi une alarme qui permettra de gérer le cas de timeout du fils. En effet, situé dans le père, cette alarme ne sera pas impactée par d'éventuelles alarmes se trouvant dans les fonctions testées.

Nous attendons ensuite la fin du fils à l'aide de la commande waitpid, et récupérons la valeur de sortie du processus que nous mettons dans l'adresse &status.
Finalement, nous annulons l'alarme à l'aide de la fonction (alarm(0)), puis récupérons le temps d'exécution à l'aide de la fonction gettimeofday() et un calcul de temps trivial.

Nous affichons ensuite la sortie du framework sur la copie du file descriptor de la sortie standard à l'aide de la fonction print_log(). Cette fonction print sur le file descriptor passé en paramètre à l'aide de la fonction dprintf(file_descriptor, "foo").

3.1.2 Gestion du mode cmd

Pour le mode cmd, nous commençons par créer une variable de type FILE*, où sera renvoyé le retour de popen(commande) en mode écriture.

Nous récupérons le file descriptor du FILE* à l'aide de la fonction *fileno(cmd_file)*. Nous passons ensuite la sortie du framework dans ce file descriptor, récupérons la valeur de "WEXITSTATUS(pclose(cmd_file))", puis nous comparons cette valeur.

Si cette dernière vaut 2 (sur fedora en tout cas), alors la fonction est une réussite, nous affichons donc le test avec "SUCCESS". Dans le cas où la commande n'a pas renvoyé un état de succès, nous affichons le test de cette fonction comme "FAILURE" et incrémentons le nombre de tests échoués par 1.

3.1.3 Gestion du mode logfile

Si le mode logfile est activé (ie. non NULL), nous créons un nouveau filedescriptor, et écrasons tout fichier avec ce nom pour le remettre à 0 avant même la création des processus fils de test.

Dans le fils, si le mode verbose est activé, nous enregistrons la sortie standard et d'erreur des tests, sinon, nous n'enregistrons rien dans ce fichier depuis le processus fils.

Dans le père, nous enregistrons la sortie du framework test par test dans le fichier indiqué à l'aide de la fonction print_log() qui prend en paramètres la structure du framework, le status du processus fils, le numéro du test, son temps d'exécution et un file descriptor.