ECOLE NATIONALE SUPERIEURE
DES MINES DE NANCY

Département : Informatique
2ème année

Conception et réalisation d'un programme « fari » ayant un comportement similaire à la command « make »

Réalisé par :

- LAI Linxue
- FOURKA Bilal

Année universitaire : 2019-2020

I. Conception de notre programme fari :

A propos de notre conception de ce projet, nous avons suivi les étapes suivantes :

1. Initialisation

Tout d'abord, on pensait à créer et initialiser les variables qu'on utilise.

2. Lecture des informations du « farifile »

On utilise la fonction : ***getline(&line_buf, &line_buf_size, fp);***

C'est pour lire chaque ligne de notre fichier de description et l'enregistrer dans une variable line_buf.

3. Classification des informations

Afin réussir à lire les informations du fichier de description, on utilise une boucle pour classifier les informations suivant les noms de fichiers qui sont lus par la fonction getline(). Selon la syntaxe du fichier de description, les informations sont distinguées par les mots-clés C, H, F et B. Quand on rencontre le mot #, on doit ignorer la ligne. Dans la boucle, la fonction strToken = strtok(line_buf," ") permet de séparer les mots dans chaque ligne par des espaces.

Dans cette partie, on appelle plusieurs fois la fonction :

struct list *getListFromLine(struct list *T,char *buff, int *nb,json_object ** jarray)

C'est pour obtenir les listes de fichiers à partir des différentes lignes. On les stocke sous forme de listes chainées à partir de la structure « **struct list** ».

4. Traitement des fichiers « .h »

On utilise la fonction : ***time_t TraiterHFile(struct list * HT)***

C'est pour parcourir la liste de fichiers et de retrouver le « **st_mtime** » du fichier « .h » le plus récent. Pour réussir cette partie on appelle la fonction stat(2) sur chaque fichier « .h » dans la fonction.

5. Compilation des fichiers « .c » aux fichiers « .o »

Ici on utilise la fonction : ***int traiterCFile(char *Cfile ,time_t stMtime)***

C'est pour comparer le temps de la dernière modification entre chaque fichier « .c » avec le stMtime du fichier « .h » qu'on a obtenu avant en utilisant la fonction **traiterHfile** ou avec celui du fichier « .o » correspondant.

Pour chaque fichier « .c », Si aucun fichier « .o » ne correspond, il faut recompiler le « .c ». Aussi, Si le fichier « .o » existe mais le fichier « .c » est plus récent que le fichier « .o » correspondant, on recompile. Et s'il existe un fichier « .h » plus récent qu'un seul fichier « .o », on doit recompiler les fichiers en utilisant la fonction :

execCmdC(command,P->file,line_buf,Fnb,FT,&jarrayCommands,&error_msg1);

Sinon on ne fait pas de recompilation.

6. L'exécution des fichiers « .o » au fichier exécutable

Ici on appelle la fonction : ***time_t getMaxMtimeO(struct list *CT)***

C'est pour parcourir la liste des fichiers « .c » et de trouver le fichier « .o » le plus récent.

S'il existe un fichier « .o » plus récent que le ficher exécutable, on doit recompiler les fichiers au fichier exécutable en utilisant la fonction

execCmdE(command,Cnb,CT,Fnb,FT,Bnb,BT,ET,&jlinking,&error_msg2);

Sinon on ne fait pas de recompilation.

7. Destruction des variables et des descripteurs.

Après avoir bien compilé tous les fichiers, on fait la destruction des variables et des descripteurs.

II. Les extensions :

1. Fichier de logs JSON :

Pour réaliser les enregistrements des traces de commande, on a créé les objets JSON au début et on a utilisé plusieurs fonctions de la bibliothèque **« json-c »** pour créer les différents champs et on a produit un fichier de logs au format JSON nommé « *logs.json* » en utilisant la fonction ***fopen("logs.json", "w")***.

Ensuite, on a conservé les informations de traces à chaque fois qu'il y a des compilations ou des exécutions qui sont faites. ***#include <json-c/json.h>*** est ajouté à notre programme pour permettre l'accès à certaines fonctions.

Après, on a écrit tous les messages nécessaires au fichier ***log.json*** sur chaque ligne et on le ferme à la fin.

Après avoir fini toutes les étapes en dessus, le fichier est rempli en respectant le format donné dans l'énoncé. Ce fichier donc est utile pour déboguer l'exécution de notre programme.

2. Compilation du Java :

Pour réaliser cette extension, on a ajouté notre fonction pour la compilation avec javac :

int execCmdJ(char **command,int Jnb,struct list *JT,json_object **jarray,json_object ** error_msg)

Lorsqu'on retrouve des lignes avec un flag J dans « farifile », on stocke les informations de fichiers .java sous forme de listes chainées. En utilisant notre fonction ci-dessus, on peut compiler les fichier .java aux fichiers « .classe ». Finalement, les fichiers « .classe » peuvent être exécuté par la JVM avec la commande ***java [basenameFile]***.

3. Entêtes liés aux fichiers sources :

Le but de cette extension est de rajouter un nouveau format pour le fichier de description « CH » qui permettra de lier un ou plusieurs fichiers entêtes à un ou plusieurs fichiers sources.

Dans notre programme on construit une liste *CH pour conserver les informations sur chaque ligne qui commence par le drapeau CH. Ensuite on utilise la fonction dont le prototype est :

void traiterCHFile(struct listCH *,int,int *,struct list *,json_object **,json_object **,int *) ;

C'est pour traiter notre fichier « .c » marqué par CH. Par exemple on a une ligne CH f1.c f2.h dans le « farifile », alors Fari devra recompiler f1.c si f2.h a été modifié. Si f1.h est modifié, le comportement ne change pas.

4. Continuation sur erreur :

En ajoutant cette extension, notre programme fari doit pouvoir continuer même s'il y a des erreurs lorsque le drapeau « -k » est passé en paramètre avec ou sans argument du nom du farifile. Pour implémenter cette extension il fallait considérer une variable errr signifiant 0 s'il n'y a pas des erreurs ou 1 sinon. Du coup, le programme continue à fonctionner en affectant 1 a cette variable en cas d'erreur. Cette variable est passé par adresse pour assurer sa modification directe.

5. Globbing :

Cette extension permet l'utilisation de « * » pour signifier tous les fichiers tels que « * » est remplacé par n'importe quel mot. Pour réaliser cette fonctionnalité, on a utilisé la fonction **glob(3)**. L'idée est d'utiliser cette fonction sur tous les noms de fichiers enregistrés, ceci ne modifie pas le fonctionnement normal du programme. On utilise la même fonction dans le « main » :

struct list *Globbing(struct list *T,int *nb);

III. Difficultés confrontées et les solutions utilisées :

- Ignorer les espaces à la fin de chaque ligne dans le « farifile »
  solution : créer une autre variable de type char * qui s'appelle **sep** dans la fonction **getLsitFromLine( )** et utiliser la fonction **strtok_r** avec ' \n' comme séparateur.
- En évoluant dans le projet, notre code était moins maintenable et difficile à lire et à comprendre même si c'est nous qui l'avons développé.
  solution : utilisation des fonctions afin de maitriser le code et diminuer la marge des erreurs générées.
- Calculer le nombre de fichiers de chaque type afin de l'utiliser. Par exemple, pour s'assurer qu'il n'y a qu'un seul fichier exécutable type « **E** ».
  solution : on a fait un passage par adresse afin de modifier la vraie variable directement (ncb++).
- Enregistrement des noms de fichiers, il fallait trouver une structure de données dynamique. Un tableau dynamique demande un parcours afin de calculer le nombre de fichiers et utiliser la fonction malloc.
  solution : on a utilisé les listes chainées ce qui nous permet d'ajouter des éléments sans penser au nombre de fichiers.
- La fonction qui traite les fichiers sources « **.c** » retourne 0 ou 1 pour savoir si on doit recompiler le fichier ou pas, du coup cette fonction ne peut pas retourner le 'st_mtime' du fichier « **.o** » le plus récent.
  solution : créer une autre fonction getMaxMtimeO( ) qui aura un seul objectif.
  C' est de calculer le 'st_mtime' du fichier .o le plus récent.
- L'installation de la librairie json-c n'était pas facile parce qu'on travaille avec des machines virtuelles. Sur les sites officiels, il n'y a pas une procédure claire pour installer json-c.
  solution : sudo apt-get install libjson0 libjson0-dev

IV. Répartition des heures :

Conception : on était en train de conceptualiser le projet ensemble pendant 5 heures repartis sur deux jours (19 et 20 octobre).

Implémentation : c'est l'étape qui nous a pris le plus de temps, approximativement 38 heures en somme.

Tests : on a essayé de tester notre programme sur de nombreuses possibilités de fichiers de description. Cette étape nous a pris 5 heures.

Rédaction du rapport : cette étape nous a pris 15 heures en somme.

	BILAL	LINXUE	Totale du binôme
Conception	5		10
Implémentation	21	17	38
Tests	**2**	**3**	5
Rédaction de rapport	6	9	15
Totale par étudiant	34	34	68

Annexes

Annexe 1 : diagramme du fonctionnement de notre programme fari