Projet fari – système de build automatique

RS : Réseaux et Systèmes

L'objectif de ce projet est de réaliser une version simplifiée d'un système de build automatique à l'aide de `stat(2)`, `fork(2)` et `execvp(3)`.

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.

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.

Informations complémentaires. Des informations complémentaires (foire aux questions par exemple) pourront être fournies sur le dépôt git projet-1920-fari du groupe RS2020 du gitlab de l'école : `https://gitlab.telecomnancy.univ-lorraine.fr/rs2020/projet-1920-fari`. 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.

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

2 Description du projet fari

Fari est un mot en esperanto qui signifie *make* en anglais `;-)`

Le projet consiste à écrire une version simplifiée du programme `make`, qui aide à automatiser la compilation de programmes. `Fari` offre cependant moins de fonctionnalités que son grand frère de la collection de logiciels GNU : il ne permet que d'automatiser la compilation d'un seul programme C en utilisant le compilateur `gcc`.

La syntaxe de `fari` est la suivante : `fari [fichier-description]`. Si le nom du fichier de description est omis, on utilisera « farifile » par défaut. Si le fichier spécifié n'existe pas, `fari` termine avec un code erreur et un message.

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).
- `# du texte` un commentaire à ignorer.
- `C <fichiers>` Le chemin d'un ou plusieurs fichiers source à utiliser.
- `H <fichiers>` Le chemin d'un ou plusieurs fichiers d'entête à utiliser.
- `E <nom>` Le chemin de l'exécutable à produire.
- `F <nom>` Les « flags » à passer à `gcc` lors de la compilation.
- `B <noms>` Les bibliothèques et autres fichiers objets à ajouter lors de l'édition de lien.

Les lignes **C**, **H**, **F** et **B** peuvent contenir plusieurs éléments (plusieurs fichiers sources, plusieurs drapeaux, *etc.*) séparés par des espaces. Il peut y avoir plusieurs telles lignes, qui peuvent être vides (aucun élément hormis la lettre). Il doit y avoir une et une seule ligne **E**. `Fari` doit terminer avec un code d'erreur et un message dans tous les autres cas.

2.2 Exemple de fichier de description

```
1  # Mon exécutable s'appelle f
2  E f
3
4  # Il y a trois fichiers sources
5  C f.c
6  C f1.c f2.c
7
8  # et un fichier d'entête
9  H f.h
10
11 # j'utilise la bibliothèque mathématiques
12 B -lm
13
14 # et je souhaite ajouter le flag de debug
15 F -g
```

2.3 Déroulement de l'exécution de `Fari`

`Fari` compile tous les fichiers `.c` en fichiers `.o` (en utilisant `gcc -c`) et réalise ensuite l'édition de liens de tous les fichiers `.o` dans l'exécutable final. Comme `make`, `fari` ne recompile que ce qui est nécessaire. L'algorithme suivant est utilisé pour décider ce qui doit être recompilé :

- Pour chaque fichier `.c`
  - Si aucun fichier `.o` ne correspond, il faut recompiler le `.c` (avec `gcc -c`, et en ajoutant les drapeaux spécifiés).
  - S'il existe un fichier `.o` mais que le `.c` est plus récent, il faut également recompiler le `.c` (de la même manière).
  - S'il existe un fichier d'entête (`.h`) plus récent que le `.o`, il faut recompiler le `.c`. Il faut recompiler **tous** les fichiers source quand un fichier d'entête est modifié, même si cet entête n'est pas utilisé.

— Si l'exécutable existe et est plus récent que tous les fichiers `.o`, il n'est pas nécessaire de recommencer l'édition de liens. Dans le cas contraire, il faut utiliser `gcc -o` en ajoutant les « flags » et les bibliothèques spécifiées.

Bien entendu, si un fichier `.c` ou `.h` spécifié n'existe pas, `fari` doit terminer avec un code d'erreur et un message. Si gcc indique des erreurs de compilation, `fari` doit également terminer avec un message d'erreur.

Dans le cas où `fari` s'exécute correctement, que les différentes phases de la compilation s'exécutent correctement et que l'exécutable a été généré, le programme doit retourner le code 0 et uniquement dans ce cas.

3 Travail à réaliser

3.1 Outils à utiliser

Vous devez bien entendu utiliser l'appel système `stat(2)` pour retrouver l'âge des fichiers et déterminer ce qui doit être compilé. Référez vous à la page de manuel pour plus de détails. Vous utiliserez le champ `st_mtime` de la structure `stat` (qui correspond au nombre de secondes entre le 1/1/1970 et la dernière modification du fichier).

Pour exécuter une commande, vous utiliserez les fonctions `fork(2)` et `execvp(3)`.Le processus père (`fari`) doit vérifier le code de retour de ses fils (`gcc`) pour détecter s'il y a eu une erreur de compilation ou non.

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 farifile du projet et tester l'ensemble.
2. Écrire la boucle principale de lecture du fichier de description. Elle pourra utiliser `getline()` et affiche chaque ligne. Pour découper la ligne selon les espaces vous pourrez utiliser `strtok`.
3. Écrire le code reconnaissant les lignes blanches. Toutes les autres lignes sont ignorées.
4. Écrire le code reconnaissant les lignes **C**. Toutes les autres lignes sont ignorées.
5. Écrire une fonction traitant une ligne **C** en la concaténant à la liste des lignes **C** déjà connues. Après avoir lu tout le fichier de description, le résultat de toutes ces concaténations est affiché.
6. Traiter les lignes **H**, les lignes **F** et les lignes **B** de la même manière que les lignes **C** (par concaténation).
7. Écrire le code reconnaissant la ligne **E** et la fonction de traitement.
8. Écrire du code générant une erreur en présence de ligne ne correspondant à rien de connu, ainsi que si le nom de l'exécutable n'est pas précisé ou si plusieurs noms d'exécutables sont donnés.
9. Écrire le code traitant les fichiers d'entête. Il parcourt la liste de tous les fichiers spécifiés, et appelle `stat(2)` sur chacun d'entre eux. Si l'un des fichiers n'existe pas, une erreur est générée. Si non, la fonction retourne le maximum des `st_mtime` rencontrés.
10. Écrire le code traitant les fichiers sources. Il parcourt la liste de tous les fichiers spécifiés, et appelle `stat` sur chacun d'entre eux. Si l'un des fichiers n'existe pas, une erreur est retournée. Si non, on considère le `.o` correspondant. S'il n'existe pas ou s'il est plus ancien que le `.c` ou qu'un fichier d'entête quelconque, afficher un message indiquant qu'il est nécessaire de reconstruire le `.o` (un printf suffit pour l'instant). Il est important de s'assurer du bon fonctionnement de cette étape.
11. Construire la commande permettant de recompiler le fichier `.c` donné (« gcc -c ... », sans oublier les flags). Dans un premier temps, on génère cette commande et on l'affiche. Lorsque cela semble bon, on ajoute les appels à `fork` et `execvp`.
12. La fonction de traitement des `.c` doit retourner le maximum des `st_mtime` de `.o` (après éventuelle reconstruction).
13. Écrire le code déterminant si l'exécutable doit être reconstruit ou non.
14. Écrire le code générant la commande à utiliser pour recompiler l'exécutable (« gcc -o ... »). Comme précédemment, on commencera par afficher cette commande avant d'ajouter les appels à `fork()` et `execvp()`.
15. Traiter les cas où gcc indique une erreur de compilation (en terminant sur un message d'erreur).

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 12, pour obtenir une meilleure note il faudra donc implémenter plusieurs des extensions présentées ci-dessous.

— **Fichier de logs JSON :** Vous pouvez produire un fichier de logs au format *JSON* nommé `logs.json`. Son format est fixé et vous devrez le respecter puisque ce fichier sera utilisé par les tests automatiques. Ce fichier devrait également vous être utile pour déboguer l'exécution de votre programme. Voici son format :

```
{
    "source":[],
    "headers":[],
    "libraries":[],
    "flags":[],
    "executable_name":"",

    "fari_error_msg":"",

    "compilation":{
        "commands":[],
        "error_msg":""
    },
    "linking":{
        "command":"",
        "error_msg":""
    },

    "fari_msg":""
}
```

En reprenant le « farifile » de la sous section 2.2, voici le fichier de logs correspondant :

```
{
    "sources":["f.c", "f1.c", "f2.c"],
    "headers":["f.h"],
    "libraries":["-lm"],
    "flags":["-g"],
    "executable_name":"E",

    "fari_error_msg":"",

    "compilation":{
        "commands":[
            "gcc -g -c f.c",
            "gcc -g -c f1.c",
            "gcc -g -c f2.c"
        ],
        "error_msg":""
    },
    "linking":{
        "command":"gcc -g -o E f.o f1.o f2.o -lm",
        "error_msg":""
    },

    "fari_msg":"Compilation terminée."
}
```

Dans ce cas précis, la compilation s'est bien déroulée, il n'y a pas de messages d'erreurs mais s'il devait y en avoir vous êtes libres de mettre les messages que vous jugerez les plus pertinents (de même pour l'attribut « fari_msg »). Vous pouvez ajouter d'autres éléments dans ce fichier de logs si vous le jugez utile mais il doit au minimum contenir les élements présentés ici.
Nous vous conseillons d'utiliser une des deux bibliothèques suivantes pour générer du *JSON* : `json-c` ou `jansson` (vous avez le choix). Ces deux bibliothèques sont présentes dans les paquets de la plupart des distributions GNU/Linux :

— libjson-c-dev et libjansson-dev pour Debian, Ubuntu et Linux Mint
— json-c-devel jansson-devel pour Fedora
— json-c et jansson pour Archlinux

Toutes les deux sont installés sur les postes GNU/Linux à l'école également.
Il faudra utiliser `#include <json-c/json.h>` ou `#include <jansson.h>`. Pour compiler votre projet il faudra utiliser `gcc -o fari fichiers.c -ljson-c` ou `gcc -o fari fichiers.c -ljansson` selon la bibliothèque que vous aurez choisie.

— **Compilation du Java :** si le fichier contient des lignes `J <fichier>`, `fari` les compile comme des fichiers source java (avec le programme `javac`). On pourra considérer qu'un « farifile » ne puisse contenir que les instructions pour compiler du C ou compiler du Java mais pas les deux en même temps.

— **Continuation sur erreur :** lorsque l'on passe le drapeau -k au programme `fari`, il ne s'arrête pas immédiatement si l'un des processus fils renvoie un code d'erreur. Au lieu de cela, il lance toutes les commandes restant à faire qui ne dépendaient pas de ce fils, et renvoie un code d'erreur seulement après leur complétion.

— **entêtes liés aux fichiers sources :** actuellement `fari` recompile tous les fichiers si un entête a été modifié même si cet entête n'est concrètement pas utilisé. 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. Ainsi la recompilation devra être plus sélective : on ne recompilera que les fichiers sources dont un des fichiers entêtes liés a été modifié. Exemple de « farifile » :

```
C f.c f2.c f1.c
H f1.h
CH f2.c f2.h
E f
```

`Fari` devra alors recompiler `f2.c` si `f2.h` a été modifié, (`f1.c` n'aura pas besoin d'être recompilé car il n'utilise pas `f2.h`). Si `f1.h` est modifié, le comportement ne change pas.

— **Globbing :** on veut pouvoir utiliser le caractère `*` (*wildcard*) partout où on est amené à lister des fichiers. Par exemple, on voudrait pouvoir utiliser le fichier de description suivant :

```
C *.c src/*.c src/*/*.c
H *.h include/*.h include/*/*.h
E mon_binaire
```

Si votre solution ne prend en compte que certaines de ces possibilités de globbing, c'est déjà intéressant.
Aide : il y a une manière simple de l'implémenter et une manière ... moins simple.

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 `rs1920-` et contenir les noms des deux étudiants du binôme (par exemple `rs1920-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

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

— Un `Makefile` compilant votre projet en créant un fichier exécutable nommé `fari`

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

6 Calendrier

Des «tests blancs» seront effectués à au moins trois reprises, aux alentours du 04/11/19, du 22/11/19, et du 06/12/19. 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 15/12/2019 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.