Rapport de projet
Pratique des systèmes d'exploitation

Quentin GLIECH

25 novembre 2015

1 Introduction

J'ai l'habitude de travailler avec Git, et ce projet n'a pas été exception. Le code, avec l'historique des commits est disponible en ligne sur un dépôt GitHub : `https://github.com/sandhose/maque`. Ayant l'historique de mes commits à l'appui, j'ai organisé ce document dans l'ordre chronologique dont j'ai codé les différentes parties du script. Je fais également référence à ces commits dans ce PDF, et ces hash de commits sont cliquable si vous voulez voir directement le commit en question. De même pour les numéros de ligne.

2 Fonctionnement global du script

2.1 Parsing des options - Commit `be137be`

Puisque c'est la première chose que fait réellement mon script, c'est également la première fonction que j'ai codé (`parse_commandline`), avec une fonction utilitaire pour afficher l'aide, `usage`.

Ma première implémentation de cette fonction utilisait `getopts`, mais je me suis rapidement rendu compte que cette commande faisait partie du projet GNU, et n'était donc pas disponible de manière standard sur n'importe quel système UNIX, y compris mon Mac sur lequel je travaillais. J'ai donc changé pour `getopt` (cf. commit `096546b`), qui est plus *POSIX Compliant*, et avait donc l'avantage non négligeable de fonctionner sous Mac OS.

L'implémentation en soi est relativement simple : un appel de `getopt`, avec une vérification qu'elle s'est bien passée, puis une itération à travers tous les arguments, avec un simple `case`, en enlevant bien les arguments au fur et à mesure, qui va définir un certain nombre de variables. Les `shift` à chaque traitement d'un argument servent pour ensuite récupérer la liste des cibles à compiler dans les arguments qui restent.

2.2 Parsing du Makefile - Commit `b634c58`

On entre la dans je pense la partie la plus intéressante du projet. Je pense qu'il y avait globalement 2 manières de fonctionner pour chercher les cibles dans le Makefile. Une qui consistait à chercher les cibles directement à coups de `grep`, et naviguer de cible en cible de cette manière ; l'autre qui était de lire le Makefile en amont ligne par ligne et sauvegarder de manière structurée dans des fichiers temporaires. J'ai choisi la deuxième méthode, qui est je pense plus stable (plus facile de trouver des erreurs de syntaxe dans le makefile avant de faire quoi que ce soit), bien que peut-être un peu plus complexe à implémenter.

J'ai donc une boucle qui va lire le fichier ligne par ligne. La première opération sur une ligne est de supprimer les commentaires (donc tout ce qui a après un `#`), et les espaces en fin de lignes, pour pour pouvoir ensuite facilement supprimer les lignes vides.

Pendant la lecture du makefile, je maintiens une variable `CURRENT_TARGET`, qui contient la cible en cours de lecture. Pour chaque ligne, je commence par vérifier si c'est une ligne commençant par un tab (avec un `printf` pour avoir littéralement le caractère tab), et j'ajoute donc cette ligne à la liste des commandes de la cible `$CURRENT_TARGET`, la cible qu'on est en train de lire *(l. 106)*. Le second test *(l. 109)* vérifie s'il s'agit d'une ligne de définition de cible. Je change donc à ce moment la variable `CURRENT_TARGET`, et je crée un dossier temporaire pour la cible, qui va contenir 2 fichiers : un fichier avec la liste des dépendances, et un fichier avec la liste des commandes. Aussi, à ce moment la, j'en profite pour définir la cible par défaut si cela n'a pas été fait auparavant (la cible par défaut correspond donc à la première cible trouvé dans le makefile). Enfin, si la ligne ne correspond à aucun de ces pattern, je considère qu'il y a une erreur de syntaxe, et arrête le script avec une erreur.

2.3 Construction d'une cible - Commit `0f171fe`

La fonction `build_targets` *(l. 200)* va quand à elle pour chacun de ces arguments appeler la fonction `build_target_if_needed`, retourner 1 si une des cibles a été construite, 0 si aucune des cibles n'a été construite, et 2 s'il y a eu une erreur quelconque lors de la construction d'une des cibles.

`build_target_if_needed` *(l. 136)*, comme son nom l'indique, va construire la cible passée en argument si nécessaire. Cette fonction commence donc par une série de tests. On teste d'abord si la cible est définie dans le Makefile. Si ce n'est pas le cas, et que le fichier existe, on se contente de retourner 0, et s'il n'existe pas, on lève une erreur.

Dans cette fonction, je maintiens une variable `NEED_BUILD`, initialisée à `false`, et qui est fixée à `true` dès qu'une condition fait qu'il faut compiler la cible. Une condition, par exemple, est si le fichier n'existe pas. Dans ce cas, on fixe d'emblée `NEED_BUILD` à `true`. Ensuite, l'on s'intéresse aux dépendances, si elle en a. On va d'abord appeler la fonction `build_targets` avec comme argument la liste des dépendances. Si cette fonction retourne 1, c'est qu'une des dependence a été construite, et que la cible doit donc l'être aussi. Si elle

retourne 2, c'est qu'il y a eu une erreur, et l'on fait remonter cette erreur en retournant 2. Enfin, on teste à l'aide de l'option `-newer` de `find` si l'une des dépendance est plus récente que la cible. Si c'est le cas, à nouveau l'on assigne `true` à `NEED_BUILD`.

Après tous ces tests, il ne reste plus qu'a vérifier la valeur de `NEED_BUILD`, et d'appeler la fonction `run_commands_in_file` avec le fichier contenant les commandes de la cible.

2.4 Execution des commandes - Commit `8f93354`

La commande `run_commands_in_file` *(l. 216)* va simplement prendre en paramètre un fichier, le lire ligne par ligne, afficher les commandes, les exécuter (si l'option `-n` n'a pas été précisée), et retourne 1 en cas d'erreur (sauf si l'option `-k` a été indiquée).

3 Autre

3.1 Messages d'erreur

J'ai voulu rendre les messages d'erreur les plus proches de `make` possible. Pas de raison particulière, à part "pourquoi pas ?". J'ai donc au début du fichier une série de fonctions pour afficher des messages correctement formatés facilement. On y trouve aussi une fonction `debug` qui permet d'afficher des messages si l'option `-v` (pour *verbose*) est indiquée. Ces messages s'affichent dans une autre couleur, et avec une certaine indentation, pour mieux visualiser les différents niveaux de récursion (cf. variable `DEBUG_INDENT` et les fonctions `indent/shift_indent`). J'avais l'intention au départ d'internationaliser ces messages, mais je n'ai pas trouvé de manière propre et standard de gérer les traductions en shell.

3.2 Compatibilité

Globalement, j'ai essayé de rendre le script le plus proche des standards POSIX possible. Par exemple, utiliser `getopt` sans extensions GNU (bien que j'aimais la possibilité de définir des options longues qu'elles offrent) plutôt que `getopts`. Au final, mon script marche autant sur Debian/Ubuntu que Arch Linux et Mac OS (c'est les 4 systèmes où j'ai testé mon script)

4 Notes personnelles

Je n'ai pas personnellement rencontré de vrai gros problèmes, et j'ai trouvé ce projet intéressant à réaliser. J'ai appris le fonctionnement de `getopt`, que j'ai pas tardé à utiliser dans d'autres scripts perso. Aussi, c'était intéressant de me plonger dans le code des autres groupes pour voir les différentes manières de faire (du genre, faire la récursion au niveau du script, et pas que d'une fonction,

ou le fait de récupérer les cibles à la volée à coups de `grep` au lieu de parser le makefile en amont, comme certain ont choisi de faire).

J'avais pas mal de fonctionnalités que je voulais encore implémenter, type l'internationalisation, le support des variables, et des cibles "génériques", et y ai pas mal réfléchi, mais comme l'indique les dates des commits, j'ai bouclé le projet relativement tôt, et n'y ai plus touché pendant plus de 2 semaines, et je ne me suis pas pris le temps de les implémenter.

Enfin, la rédaction de ce rapport m'a permis de réutiliser un peu LaTeX, que j'avais jamais utilisé uniquement en CLI, et m'a donc permis de mettre en place un environnement sympa pour éditer du LaTeXdans vim, avec rendu en temps réel.