Université de Strasbourg

Licence d'informatique – Semestre 3

Luigi Coniglio

Pratique des systèmes d'exploitation - 2015/1016

Projet - réalisation de l'outil 'make'

Étudiant:	Luigi Coniglio
E-Mail:	luigi.coniglio@etu.unistra.fr

Table of Contents

NB: Toute référence à l'outil "make" dans ce rapport fait référence à la **version 3.81** de GNU Make.

1 Présentation de 'mike'

Le script *mike.sh* représente une version réduite et plus souple du bien plus connu utilitaire *'make'*. En étant une version réduite, *mike.sh* n'accepte qu'une liste restreinte des options disponibles dans *'make'* et une syntaxe très élémentaire dans le makefile (qui peut-être appelle exceptionnellement: '*makefile*', '*Makefile*', '*mikefile*' ou '*Mikefile*'). En dehors de ces limitations 'mike' se comporte de façon presque identique à son grand frère 'make'. Pour plus d'informations à propos des options disponibles vous pouvez exécuter la commande `./mike.sh –help` .

2 Illustration de l'algorithme

La structure arborescente du fichier "makefile", où l'exécution des commandes de chaque cible est conditionné par la mise à jour de ses dependences qui sont aussi des cibles, suggère beaucoup une solution du type récursif.

Une cible est traitée seulement si au moins une de ses dépendances a été traitée plus récemment que la cible, en plus (s'il est nécessaire) chaque dépendance doit être mise à jour avant le traitement de la cible. Lors de l'exécution des commandes d'une cible il faut donc traiter d'abord ses dependences comme des cibles pour les mettre à jour.

Exemple de pseudo-code:

```
Fonction traitement ( C )
        Pour tout D dans dépendances de  C  faire
                Si D est une cible
                        traitement ( D )
                Fin Si
        fait
        Pour tout D dans dépendances de C faire
                Si D est plus récent que C
                        executer ( C )
                Fin Si
        fait
Fin Fonction
```

Cette première version très simplifiée de l'algorithme synthétise bien le principe de fonctionnement de 'mike.sh', par contre il ne prend pas en compte quelque cas particulier, comme par exemple le cas d'une cible n'ayant aucune dépendance ou d'une cible (ou dépendance) qui n'est jamais créé par l'exécution de ses commandes (comportement proche a ceux des cibles du type .PHONY utilisee par l'outil 'make').

La fonction appelée "traitement" dans le pseudo-code est représenté par la fonction `build` dans "mike.sh".

3 Principales difficultés

3.1 Extraire les dépendances et les règles d'une cible

Réussir à extraire sans aucun problème les dépendances et les règles d'une cible était la première difficulté que j'ai rencontrée. Le problème est causé principalement par deux raisons:

1. Une éventuelle présence de commentaires et lignes vides dans le makefile.
2. La présence éventuelle de caractères spéciaux par rapport aux expressions régulières dans les noms des cibles et des dépendances qui, avec `sed`, peut causer des résultats inattendus.

Solution:

1. Travailler sur un fichier temporaire `/tmp/$(basename $0).$$` qui contient une version préprocessée du fichier 'makefile', n'ayant ni des lignes vides ni des commentaires[1].
2. Utiliser une fonction pour échapper tous les caractères spéciaux.[2]

3.2 Préservér les valeurs des variables lors des appels récursifs

Dans la fonction `build` la variable `DEPS` contient la liste de toutes les dependences de la cible en cours de traitement. Lors des appels récursifs de cette fonction le contenu de `DEPS` est écrasé par la liste des dependences de la nouvelle cible en cours de traitement.

J'ai trouvé trois solutions possibles à ce problème:

1. Régénérer la liste des dependences après avoir appelé build recursivement
2. Specifier `DEPS` comme une variable locale. De cette façon les changements apportés à `DEPS` ne seront visibles que dans la fonction où ils ont été faits.

[1] Voir la fonction `preprocess`
[2] Voir la fonction `escape_bre`

3. Exécuter la fonction build dans une *subshell.* Une *subshell* est un nouveau processus du shell qui reçoit les attributs du shell parent (y compris les variables, même s'elles n'ont pas été exportés).
   Étant un nouveau processus il ne peut pas modifier la valeur de ces variables dans le shell parent: *«Builtin commands grouped into a (list) will not affect the current shell.»*[1] La syntaxe pour exécuter une commande dans une subshell est la suivante: (commande) .

3.3 Lires et exécuter les commandes ligne par ligne

La commande sh -c "$CMDS" peut-être utilisée pour exécuter plusieurs commandes en une seule fois, par contre *"mike.sh"* doit être capable de s'arrêter si une commande termine son exécution avec un code de sortie different de 0. On a donc besoin d'exécuter chaque commande séparément et vérifier s'il y a eu quelque problème dans l'exécution et pour faire cela il faut lire ligne par ligne la liste des commandes.
Une façon très utilisé pour lire une variable ligne par ligne est:

```
printf '%s\n' "$CMDS" | while read -r cmd
do
   sh -c "$cmd"
done
```

La commande read lit une ligne depuis l'entrée standard et copie son contenu dans la variable cmd. Par contre il y a un cas particulier pas mal caché qui nous empêche d'utiliser cette solution: notre makefile pourrait bien avoir une commande qui utilise l'entrée standard.
Exemple:

```
Ecrire_fichier:
      echo "Ecrivez quelque chose"
      cat > fichier
```

Dans cet exemple la ligne cat > fichier se traduit par une lecture depuis l'entrée standard effectué par l'outil cat. Dans la boucle while du premier exemple par contre l'entrée standard est représenté par la sortie standard de la commande printf, l'utilisateur n'aura donc aucune possibilité de saisir du texte car printf aura déjà tout fait pour lui.

Solution

[1] Manuel de *Debian Almquist shell* (DASH)

Dans le shell le contenu de la variable IFS est utilisé pour définir les séparateurs de champ reconnus par l'interpréteur, d'habitude il contient l'espace, la tabulation et le retour de ligne. Une boucle for utilise ce séparateur pour évaluer une liste de valeurs. Dans notre cas les lignes des commandes sont séparées par un retour de ligne, il nous suffit de changer le contenu de la variable IFS avec rien d'autre que le retour de ligne et utiliser une boucle for pour traiter les commandes une par une.
Attention: on utilisera le mot-clé local car on ne veut pas que la valeur de IFS soit changée pour toute la suite de notre programme.

```
fonction (){
      …
      local IFS='
'
      for cmd in $CMDS
      do
            sh -c ''$cmd''
      done
      …
}
```

Voir la fonction execute dans le programme.

NB: l'outil make ne considère pas le caractère ; comme un séparateur de champ.
On considère par exemple cette ligne de commande dans un makefile:

```
cat fichier_inexistant ; echo "*** fin fichier ***"
```

bien que la commande cat n'ait pu s'exécuter avec succès, la commande echo sera toujours exécutée car elle se trouve dans la même ligne.

3.4 Gérer les dependencies quand l'option -k est activé

L'outil make permet à l'utilisateur de specifier l'option -k pour continuer l'exécution même si une commande retourne une erreur. Lors d'une erreur make (de façon intelligente) continue seulement la construction des cibles qui ne sont pas liées à la dépendance où il y a eu l'erreur. Si on considère le makefile suivant:

```
toto: titi tata
      diff titi tata > toto
titi: foo
      cat foo  >> titi
foo:
      mv fichier_inexistant foo
tata:
      who > tata
```

L'exécution de la commande make toto n'aura aucun effet à cause de l'erreur dans la commande mv utilisé pour générer le fichier foo. Par contre lors de l'exécution de make -k toto le processus ne s'arrêtera pas, bien que l'erreur soit toujours présente, en omettant les cibles qui ont besoin de foo pour être créé. De cette façon foo, titi et toto ne seront pas générés, par contre tata, n'ayant aucune liaison avec le fichier foo, sera traité normalement.

Solution

Le script *mike.sh* doit être capable de suivre la même logique d'exclusion des cibles affectées par une dépendance qui ne peut pas être mise à jour. Pour ça j'ai pensé à utiliser un fichier temporaire *"BROKENDEPS"* (/tmp/brkdp.$$) où sont stocké les noms de toutes les cibles qui ont eu une erreur de traitement.
Le mécanisme peut être synthétisé par 2 règles (à utiliser seulement lors de l'activation de l'option -k) ;

1. Toutes les fois qu'une cible ne peut pas être mise à jour, son traitement est interrompu et son nom est ajouté au fichier *"BROKENDEPS"*
2. Avant de traiter une cible, si le nom d'une de ses dépendances figure dans le fichier *"BROKENDEPS"*, la cible ne sera pas être mise à jour.

3.5 Dépendances répétées

3.5.1 Simple répétition

Un makefile peut parfois contenir des dependences répétées, c'est-à-dire de cibles qui sont appelées plusieurs fois dans le même processus de compilation.
Exemple:

```
etudiant: professeur ecole
	suivre_cours professeur > etudiant

professeur: ecole
	enseigner_dans ecole > professeur

ecole:
	construir ecole
```

Dans cet exemple lors de l'exécution de make étudiant la dependence école se répète deux fois, mais: make ne traite pas la dépendance plus d'une fois, heureusement on n'aura pas deux écoles toutes les fois qu'on souhaite faire suivre un cours a un étudiant!

3.5.2 Makefiles circulaires

Les makefiles circulaires représentent un cas particulier des dependences répétées. Un makefile est circulaire quand une de ses cibles contient elle-même dans son arbre de dépendances. Exemple:

```
gnu_project: contributors
	echo "GNU's Not Unix !"
contributors: gnu_project
	svn checkout svn://gcc.gnu.org/svn/make
```

L'outil make est capable de prévenir la création d'un loop infini (qui est parfois bien caché dans des makefiles très riches) en éliminant les dependences qui ont le même nom que l'une des cibles en cours de traitement.

3.5.3 Mike.sh et le dépendances répété

Pour implanter le même principe (soit les répétitions simples que les problèmes de circularité) *mike.sh* utilise une liste qui est mise régulièrement à jour lors du traitement des cibles. Toute cible déjà traité (qui appairait dans la liste) ne doit pas être traité une deuxième fois. Mike affiche en plus un message de warning qui indique quelle dépendance est répétée et depuis quelle cible parente elle a été appelé.

4 Extras

Ici une liste des petits extras que j'ai rajouté par rapport au sujet du projet :

- Cible par défaut: *mike.sh* utilise la première cible du fichier makefile si aucune cible n'a pas été spécifiée (comme l'outil make).
- Syntaxe moins rigoureuse: *mike.sh* accepte les cibles et les commentaires mêmes s'ils commencent par un nombre quelconque des espaces.
- Option -c <répertoire>: pour changer de répertoire avant de lire le makefile et exécuter les commandes (très utiles pour des appelles récursives dans différents répertoires).
- Option -s: mike n'affiche pas les commandes quand elles sont exécutées
- Option –help: affiche des informations à propos de l'utilisation de mike.sh