Projet SimpleMake

RS : Réseaux et Systèmes
Deuxième année

L'objectif de ce projet est d'écrire une version simplifiée de `make(1)` grâce à `stat(2v)`, `fork(2)` et `execvp(3)`.

Évaluation de ce projet

Vous devez rendre un mini-rapport de projet (3 pages maximum), une archive `tar` contenant votre source ainsi qu'un makefile permettant de compiler votre projet. L'archive ne devra contenir aucun fichier objet (pas de fichier compilé).

- **12 points** seront attribués de manière semi-automatique par évaluation des fonctionnalités de votre programme. Après l'avoir compilé, nous utiliserons votre programme pour compiler différents projets de test. Vous perdrez des points si votre programme ne se comporte pas comme attendu. Pour le bon fonctionnement de ce processus (et l'attribution des points correspondants), votre programme doit renvoyer un code d'erreur de 0 en cas de succès et seulement dans ce cas. La note *maximale* d'un projet **ne compilant pas en l'état** sur neptune est 8.
- **4 points** seront attribués en fonction de la qualité (subjective) du source de votre programme.
- **4 points** récompenseront la qualité du mini rapport que vous joindrez dans votre archive. Vous y détaillerez 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, codage, tests, rédaction du rapport).

La tricherie sera **sévèrement punie**. Voir `http://graal.ens-lyon.fr/~mquinson/teaching.html#triche`

1 Description de SimpleMake

Vous devez écrire un programme `SimpleMake`. Comme le programme `make`, `SimpleMake` aide à automatiser la compilation de programmes. `SimpleMake` offre cependant moins de fonctionnalités que son grand frère : il ne permet que de compiler un seul programme C et suppose que vous utilisez le compilateur `gcc`.

La syntaxe de `SimpleMake` est la suivante : `SimpleMake [fichier-description]` Si le nom du fichier de description est omis, on utilisera «`smakefile`». Si le fichier spécifié n'existe pas, `SimpleMake` termine avec un message d'erreur.

1.1 Syntaxe des fichiers de description

Chaque ligne du fichier de description peut être l'une des choses suivantes :

- Une ligne blanche (ne contenant que des caractères blancs tels que des espaces et des tabulations).
- C <fichiers> La spécification d'un fichier source à utiliser.
- H <fichier> La spécification d'un fichier d'en-tête à utiliser.
- E <nom> La spécification du nom de l'exécutable à produire.
- D <nom> ou F <nom> La spécification des drapeaux («flags») à passer à `gcc` lors de la compilation.
- B <nom> La spécification des bibliothèques et autres fichiers objets à ajouter lors de l'édition de liens.

Les lignes **C**, **H**, **D** (ou **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 vide (aucun élément hormis la lettre).

Il doit y avoir une et une seule ligne **E**. `SimpleMake` doit terminer avec un message d'erreur dans tous les autres cas.

1.2 Actions réalisées par `SimpleMake`

`SimpleMake` 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`, `SimpleMake` 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'en-tête plus récent que le `.o`, il faut recompiler le `.c`. Il faut recompiler *tous* les fichiers source quand un fichier d'en-tête est modifié, même si cet en-tête 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 drapeaux et les bibliothèques spécifiés.

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

2 Réalisation

2.1 Outils à utiliser

Vous devez bien entendu utiliser l'appel système `stat(2v)` 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 `execv(3)`. Les arguments passés à la commande doivent être sous la forme d'un tableau de chaînes. Le processus père (`SimpleMake`) doit vérifier le code de retour de ses fils (`gcc`) pour détecter s'il y a eu une erreur de compilation ou non.

2.2 Stratégie possible

Si vous pouvez, travaillez sans utiliser ces conseils. Si vous êtes bloqués, voici comment j'ai procédé :

1. Écrire le code pour trouver le nom du fichier de description. Écrire le makefile du projet et tester l'ensemble.
2. Écrire la boucle principale de lecture du fichier de description. Elle utilise `getline()` et affiche chaque ligne.
3. Écrire le code reconnaissant les lignes blanches.
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 **D** ou **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'en-tête. Il parcourt la liste de tous les fichiers spécifiés (avec strtok(3) ?), et appelle `stat` 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'en-tête 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 un fichier `.c` donné («gcc -c ...», sans oublier les drapeaux). Dans un premier temps, on génère cette commande dans un tampon et on l'affiche. Lorsque cela semble bon, on la découpe (avec strtok ?) puis 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).
16. Apporter les touches finales au code, le nettoyer et le tester convenablement.