TP2 : Bibliothèques

1 Objectifs

- ➢ Utiliser les différents types de variables de make.
- ➢ Savoir créer un makefile avec des règles implicites.
- ➢ Expérimenter la construction et l'utilisation de bibliothèques statiques et dynamiques.
- ➢ Expérimenter la notion de plugin en C.

2 Variables de Make

2.1 Utilisation de variables

La première partie de ce TP est la suite du TP précédent.

① Dans le répertoire de l'application pacman préserver le makefile final de la scéance précédente.
Créer un nouveau makefile sur la base de ce makefile ainsi sauvegardé.

② Dans cette nouvelle version utiliser intensément les variables suivantes pour écrire les règles :
CC = gcc
CFLAGS = -c
MOVE = /bin/mv -f
RM = /bin/rm -f
BIN = $HOME/bin
OBJS = "à remplacer par la liste des .o"

③ Ajoutez et utilisez des variables
- ➢ `EXE` : pour le nom de l'exécutable,
- ➢ `INCLUDES` : le chemin des "header files" des bibliothèques utilisées (-I . . .),
- ➢ `LIBS` : le nom des fichiers de bibliothèques (-lm . . .)
- ➢ `LDFLAGS` : qui doit décrire les options de reliure, notamment le chemin des libs (-L . . .)

④ Rendre l'ajout de l'option « -g » aux FLAGS (`CFLAGS` et `LDFLAGS`) conditionnel à la définition de la variable DEBUG.
- ➢ Testez en positionnant la variable `DEBUG` sur la ligne de commande.

⑤ Définissez une variable `SOURCES` qui contient la liste de tous les fichiers du répertoire courant qui se terminent par *.c.
Indication : Utiliser la fonction `shell` ou la fonction `wildcard`

⑥ Définissez maintenant `OBJS` comme *"tous les fichiers de* `SOURCES` *transformés en .o."*

2.2 Utilisation de variables automatiques et de règles implicites

① Modifiez le makefile pour utiliser au maximum les variables automatiques ($@ , $^ , $< , . . .).

② Supprimez (ou commentez) les règles explicites de création des trois fichiers ".o". Le makefile fonctionne-t-il toujours ? Expliquez !

③ Que se passe-t-il si on fait un `touch` sur un des .h ? Expliquez le phénomène et corrigez le problème par exemple avec `makedepend` .

④ Ajoutez les 2 lignes suivantes avant les règles :

```
.SUFFIXES :
.SUFFIXES : .c .o
```

Que se passe-t-il maintenant et pourquoi ?

⑤ Créez (par fonction) une variable `ENTETES` qui contient tous les fichiers .h du répertoire courant (même principe que pour la variable `SOURCES`).

⑥ Créez une cible `print` dont les dépendances sont les fichiers ENTETES et SOURCES et qui va afficher la liste des fichiers modifiés depuis sa dernière exécution (utilisez pour cela la bonne variable automatique et un fichier vide du nom de print qui va servir de marqueur de date).

⑦ Notez bien qu'à la place de l'affichage, on pourrait utiliser une commande d'impression et que cela permettrait de réimprimer que ce qui a changé depuis la dernière impression !

3 Création de règles implicites pour la création de documents

① Récupérez et décompactez l'archive testLatex.tgz dans un autre répertoire.

② Créez un ensemble de règles implicites pour :

➢ Construire automatiquement des fichiers .eps à partir de fichiers .fig (ces derniers ont été créés avec l'utilitaire xfig)

+ La commande à utiliser est de la forme : fig2dev -L eps toto.fig toto.eps

➢ Construire 2 règles implicites construisant des fichiers `.ps` à partir de fichiers `.tex` (ces derniers sont des sources latex)

+ Il faut passer pour cela par l'intermédiaire d'un fichier .dvi créé par la commande latex. Il faut donc utiliser les deux commandes suivantes :

```
latex monfichier.tex
dvips monfichier.dvi -o monfichier.ps
```

③ N'oubliez pas de mettre à jour .SUFFIXES en conséquence.

④ Testez le tout avec les fichiers fournis en fabriquant le document ps.

4 Bibliothèque

4.1 Bibliothèque : construction statique

① Récupérez et décompactez l'archive `conversionBiblio.tgz` dans un nouveau répertoire dédié à cette partie du TP.

② Compilez séparément les fichiers `convert1.c` et `convert2.c`

③ Construisez une archive nommée `libconvert.a` à partir des deux fichiers .o générés.

④ Consultez le contenu de l'archive et vérifiez la présence des deux fichiers.

⑤ Placez l'index dans cet archive avec la commande `ranlib`.

⑥ Obtenez la liste des symboles avec la commande `nm` .

⑦ Compilez séparément le fichier `outils.c`.

⑧ Construisez un exécutable nommé `convertir` en un appel à `gcc` qui :

- ➢ compile le fichier `convert2sens.c`
- ➢ relie le fichier `outils.o`
- ➢ relie statiquement la bibliothèque `libconvert.a` située dans le répertoire courant.

⑨ Vérifiez le fonctionnement de l'exécutable produit.

⑩ Utilisez la commande `file` pour obtenir des informations sur l'exécutable. Que remarquez-vous ?

❶ Reprenez la construction de l'exécutable en forçant toutes les reliures à être statiques.

❷ Vérifiez avec la commande `file` et observez la taille de l'exécutable. Expliquez le phénomène.

4.2 Bibliothèques : construction dynamique

① Décompactez la même archive `conversionBiblio.tgz` dans un nouveau répertoire pour faire les manipulations dynamiques.

② Compilez séparément les fichiers `convert1.c` et `convert2.c` comme vous avez l'habitude de le faire.

③ Observez les fichiers .o produits avec la commande `file` . Quelle information est pertinente vis-à-vis de la reliure dynamique ?

④ Est-il nécessaire de compiler les fichiers .o d'une manière spéciale sous Linux pour pouvoir utiliser la reliure dynamique ?

⑤ Construisez une bibliothèque partagée nommée `libconvert.so` à partir de 2 fichiers .o.

⑥ Utilisez la commande `nm` pour obtenir des informations sur les symboles de cette bibliothèque. Que remarquez-vous de particulier ?

⑦ Créez un répertoire `lib` dans votre homedir et déplacez la bibliothèque dans ce répertoire.

⑧ Construisez un exécutable nommé `convertir` en un appel à `gcc` qui :

- ➢ compile `convert2sens.c` et `outils.c`
- ➢ se relie dynamiquement à la bibliothèque `libconvert.so` (utilisez les options -l et -L)

⑨ Exécutez le programme résultant. Que se passe-t-il ?

⑩ Utilisez la commande `ldd` pour afficher les dépendances de l'exécutable vis-à-vis des bibliothèques partagées. Que constatez-vous ?

❶ Positionnez la variable `LD_LIBRARY_PATH` correctement, puis exécutez `ldd` pour vérifier la résolution des dépendances, et enfin exécutez le programme.

4.3 Évolution de la bibliothèque dynamique

① Construisez la même bibliothèque dynamique que précédemment, mais :

- ➢ spécifiez que le soname interne de la bibliothèque est `libconvert.so.1` à l'aide de l'option -W
- ➢ le nom du fichier généré devra être `libconvert.so.1.0.1`

② Déplacez la bibliothèque dans votre répertoire `lib` et créez les deux liens symboliques pour que les raccourcis nécessaires à la gestion des versions soient en place.

③ Reconstruisez un exécutable du programme de conversion en le liant à cette bibliothèque avec l'option `-lconvert`.

④ Observez avec la commande ldd comment les dépendances sont affichées. Qu'en déduisez-vous sur l'utilisation des liens symboliques précédents ?

⑤ Faites une petite modification dans le fichier `convert1.c` (modifiez la chaîne utilisée dans printf pour que la chose soit visible à l'exécution) et générez une nouvelle bibliothèque avec le même soname, mais avec le nom de fichier `libconvert.so.1.0.2`.

⑥ Déplacez cette nouvelle bibliothèque dans votre répertoire `lib`.

⑦ Que faut-il faire pour que cette nouvelle version soit prise en compte par le programme de conversion sans le recompiler ? Faites-le.

⑧ Faites maintenant une nouvelle modification du même genre dans le fichier `convert1.c`.

⑨ Générez une nouvelle bibliothèque mais spécifiez `libconvert.so.2` comme soname interne.

⑩ Que faut-il faire pour que l'exécutable de conversion fonctionne avec cette nouvelle bibliothèque ?

5 Bibliothèque chargée au "runtime"

① Récupérer l'archive : `SharedPlugin.zip`

② Analyser et expliquer pourquoi on peut affirmer que la bibliothèque est liée au "runtime".