TP AISE — Compilation & ELF

Exercice : Première Bibliothèque

Écrire une bibliothèque qui expose les fonctions suivantes:

- `mylib_puts()`, qui prend en paramètre une chaîne de caractères et l'affiche sur la sortie standard. (reposer sur l'appel "`puts`" existant)
- `mylib_nbchars()`, qui affiche le nombre de caractères affichés par `mylib_puts`() depuis le début du programme.

Construire une **bibliothèque `libcustom` statique (.a) et dynamique (.so)** via un Makefile. Écrire ensuite un code de test simple qui utilise ces deux fonctions. Que se passe-t-il lorsque vous compilez contre la version statique ? et la version dynamique ? Y-a-t-il une différence au moment de lancer le binaire ? Vous pouvez utiliser des outils comme `readelf` ou `ldd` pour analyser le contenu de votre binaire.

Exercice : Deuxième Bibliothèque

Implémenter une seconde bibliothèque, qui implémente les mêmes fonctions, mais qui, en plus de compter le nombre de caractères, inverse la chaîne à afficher. On construira une bibliothèque **`libcustom2` dynamique.** Compiler le même test que précédemment contre les deux versions dynamiques en ajoutant **`"-lcustom -lcustom2"`** à la compilation. En changeant l'ordre des flags de compilation, faites en sorte que la fonction invoquée change.

Renommer maintenant votre fichier **libcustom2.so** de manière à ce que vos deux bibliothèques aient le même nom (on les mettra dans des répertoires différents pour des raisons évidentes). Recompiler le test avec **`"-lcustom"`**. Comment arranger l'ordre des bibliothèques pour choisir celle qui fournira le symbole ?

Exercice : Ma première interposition

Écrire une bibliothèque qui définit une fonction `void * malloc(size_t s)` et qui affiche sur la sortie standard la taille de l'allocation ainsi que son adresse. On n'oubliera pas bien sûr d'appeler aussi la vraie fonction de la bibliothèque standard. Écrire un test qui appelle cette fonction et compiler avec la commande suivante : "`gcc <fichier> -o test`". Charger ensuite votre bibliothèque sur le binaire **sans le recompiler.**

Exercice : Compilation manuelle

Compiler votre fichier de test sans utiliser GCC. Pour un environnement GNU, vous avez à votre disposition:

- une commande de préprocesseur : cpp
- Un compilateur: cc1
- Un assembleur: as
- Un linker: ld

Relisez bien le cours, s'il vous manque des éléments, n'hésitez pas à demander. Sachez que certains de ces programmes/ressources ne sont pas dans votre PATH par défaut. Ils sont installés dans les préfixes de GCC (souvent `/usr/lib/gcc/` ou `/usr/lib/<uname>-linux-gnu`).

Exercice : Gestion efficace de son travail

Créer un répertoire su Github (ou autre) pour sauvegarder vos tentatives/corrections. Initialiser un répertoire Git et, pour les quatre exercices précédents, écrire les makefiles associés pour dérouler la correction avec un simple "`make all`". Créez un commit pour chaque correction que vous enverrez ensuite sur le serveur de votre choix.

Exercice : mon propre `readelf`

Pour les plus avancés, écrire un programme C qui reproduit les capacités de "readelf". Le but ici n'est pas d'invoquer **readelf** mais de parcourir le programme (courant ou mappé depuis le système de fichiers) pour en analyser le contenu. Voici une rapide liste des fonctionnalités proposés:

- Déterminer si le fichier passé en paramètre est un fichier ELF
- Lister les sections présentes dans le fichier
- Listes les en-têtes de programmes du fichier
- Lister les symboles du programme (juste leur nom)
- ...

Il est fortement conseillé de s'appuyer sur `<elf.h>`, pour obtenir les types. Il est aussi possible d'utiliser libelf pour parcourir le binaire grâce à l'API, mais pour les plus débrouillards, le challenge sera de s'en passer. (En cas de besoin, s'adresser à http://www.skyfree.org/linux/references/ELF_Format.pdf)

Have fun :)