ESIR-SYS1 TP Bonus Assembleur: Compilation multi-fichiers, entrée standard, et make (Note Bonus)

François Taïani (`ftaiani.ouvaton.org`)*

1 Préliminaires

1.1 But du TP

Ce TP étend le TP 6-7-8 sur le chiffre de Beaufort. L'objectif est de factoriser l'opération de chiffrement dans une fonction, puis de partitionner le code du programme en plusieurs fichiers sources (produisant plusieurs fichiers objets), d'ajouter une lecture interactive sur l'entrée standard, et enfin d'utiliser make pour compiler le projet.

1.2 Point Bonus

La réalisation de ce TP permet de gagner jusqu'à 2 points bonus sur la note de TP assembleur.

2 Travail demandé

2.1 Partie IV : Factorisation dans une fonction (`step_4_beaufort_cipher_function.asm`)

En repartant du code `step_3_beaufort_cipher.asm` réalisé pour le TP 6-7-8, isolez dans une fonction le code permettant de chiffrer une chaîne de caractères.

Pour procéder par étape, commencez par ne passer en paramètre que la chaîne de caractères à chiffrer, en utilisant deux registres (un pour l'adresse de la chaîne, l'autre pour sa longueur). Introduisez plusieurs chaînes dans votre programme, et vérifiez que votre fonction est bien capable de chiffre ces différentes chaînes.

Dans un second temps, ajoutez la chaîne de clé comme paramètre à votre fonction, et vérifiez de nouveaux que vous êtes capable de chiffrer différentes chaînes avec différentes clés en utilisant la même fonction.

Conseils et remarques :

- En plus de `rax`, `rbx`, `rcx`, `rdx`, `rsi`, et `rdi`, donc nous avons parlé en cours, un processeur x86-64 possède 8 registres généraux supplémentaires (appelés `r8` à `r15`), que vous pouvez utiliser pour vos passages de paramètres.
- Dans la mesure où vous n'utilisez pas la pile pour stocker des variables locales à votre fonction, vous n'avez pas besoin d'insérer de prologue ou d'épilogue pour gérer `rsp` et `rbp`.
- Si besoin, pensez à sauvegarder sur la pile les registres dont vous voulez conserver les valeurs au travers de l'appel à votre fonction.

*francois.taiani@irisa.fr

2.2 Partie V : Découpage en deux fichiers (`step_5_cipher_function_only.asm` et `step_5_start_only.asm`)

Découpez maintenant le code de la partie IV en deux fichiers :

- `step_5_cipher_function_only.asm` ne doit contenir qu'une section texte (`SECTION .text`) avec le code de votre fonction. Pour que cette fonction soit utilisable à l'extérieur de votre fichier, vous devez y ajouter une directive `GLOBAL votre_fonction`, où `votre_fonction` est le nom de l'étiquette correspondant au point d'entrée (l'adresse de début) de votre fonction.
- `step_5_start_only.asm` doit contenir le reste de votre programme, sans le code de la fonction de chiffrement. Pour pouvoir utiliser votre fonction sans l'avoir définie, vous devez annoncer ("déclarer") à `nasm` que son code (sa "définition") existe à l'extérieur de ce second fichier en y insérant la directive `extern votre_fonction`, où `votre_fonction` est le nom donnée à votre fonction dans `step_5_cipher_function_only.asm`.

Compilez chacun des deux fichiers en un fichier objet (`*.o`) :

```
$ nasm -felf64 step_5_cipher_function_only.asm
$ nasm -felf64 step_5_start_only.asm
```

Vérifiez que les deux fichiers objet ont bien été créés :

```
$ ls step_5_*.o
step_5_cipher_function_only.o  step_5_start_only.o
```

En utilisant l'outil `nm` vu en cours, vérifiez ensuite que le symbole `votre_fonction` est bien un symbole de texte (référençant une fonction) défini et externe (lettre `T`) dans le fichier `step_5_cipher_function_only.o`, et que c'est bien un symbole non défini (*undefined*) et externe (lettre `U`) dans `step_5_start_only.o`. (Les autres sorties dépendront des autres étiquettes (symboles) utilisés dans vos fichiers.)

```
$ nm step_5_cipher_function_only.o
0000000000000000 T votre_function
...
$ nm step_5_start_only.o
                 U votre_function
...
0000000000000000 T _start
```

L'on peut voir ces deux fichiers objet comme deux briques qui s'emboîtent : `step_5_start_only.o` a besoin d'un symbole `votre_function` pour fonctionner, que fournit justement le fichier `step_5_cipher_function_only.o`

Il reste maintenant à "lier" ces deux fichiers objets pour créer un exécutable, en utilisant l'éditeur de lien `ld` :

```
$ ld step_5_start_only.o step_5_cipher_function_only.o -o step_5_executable
```

où `step_5_executable` est le nom que vous souhaitez donner à votre exécutable (`a.out` par défaut si vous n'indiquez pas de nom avec l'option `-o`).

Vérifiez que votre nouvel exécutable fonctionne de façon identique à celui de l'étape IV.

2.3 Partit VI : Lecture sur l'entrée standard (`step_6_message_from_stdin.asm`)

2.3.1 Préliminaires

Nous allons maintenant utiliser l'appel système `read` (ou `sys_read`) pour obtenir de l'utilisateur ou utilisatrice la chaîne à chiffrer. Commencez par vous renseigner sur cet appel système en lisant sa documentation avec `man 2 read`. `read` n'est pas à proprement parler l'appel système, mais une fonction C rudimentaire qui contient l'appel système proprement dit. Sa signature nous renseigne cependant sur les paramètres d'entrée et de sortie de `read` :

```
ssize_t read(int fd, void *buf, size_t count);
```

Vous pouvez considérer `ssize_t` et `size_t` comme des entiers de type `int`, et que l'ensemble des paramètres seront passés au noyau par des registres 64 bits.

Pour savoir quels registres utiliser pour réaliser cet appel système, et comment obtenir sa valeur de retour, consultez la section *A.2.1 Calling Conventions* du document *System V Application Binary Interface AMD64 Architecture Processor Supplement* (aussi connu sous l'acronyme *ELF x86-64-ABI psABI*), disponible sur le Web.

L'appel système `read` porte le numéro 0. Le descripteur de fichier (*file descriptor*) correspondant à l'entrée standard est aussi 0. Nous allons utiliser le fait que lorsque `read` lit depuis terminal (ce qui sera le cas ici), `read` retourne quand l'utilisateur ou utilisatrice revient à la ligne (avec la touche `Entrée`), ou indique une fin de fichier (`End of File`, `EOF`) en utilisant `Control+D`. Notez que le caractère de retour à la ligne (`newline`) sera inclus dans la chaîne renvoyée au programme.

2.3.2 Réservation d'un espace mémoire non initialisé

`read` attend un pointeur (une adresse) vers un espace mémoire dans lequel stocker la chaîne lue. Nous allons réserver une espace de 1024 octets, mais sans l'initialiser. Ce type de donnée doit être déclaré dans `nasm` dans une section spéciale appelée `bss` (pour des raisons historiques), en utilisant le mot clé `resb` (*reserve byte*). Le code suivant réserve par exemple 2 octets sans les initialiser, et définit l'étiquette (ou symbole) `two_bytes` comme une adresse pointant au début de la zone réservée.

```
SECTION .bss
two_bytes: resb 2
```

2.3.3 Appel de `read`

Une fois la zone de 1024 octets réservées, utilisez l'appel système `read` pour demander à l'utilisateur ou utilisatrice quelle est la chaîne à chiffrer, puis imprimez le résultat du chiffrement. Utilisez la fonction de chiffrement de la partie V sans la recompiler, en la liant simplement à votre programme principal au moment de l'édition de lien. Votre compilation doit ressembler à ceci :

```
$ nasm -felf64 step_6_message_from_stdin.asm
$ ld step_6_message_from_stdin.o step_5_cipher_function_only.o -o step_6_executable
```

Après avoir ajouté des messages d'invitation à destination de l'utilisateur ou utilisatrice, une exécution de votre programme devrait ressembler à cela :

```
$ ./step_6_executable
Enter the string to be encrypted/decrypted:
HELLO YOU! HOW DO YOU DO?
The encrypted/encrypted message is:
WEPCM FUG! GMH FM PMJ FM?
```

Pour finir faîtes boucler votre programme pour qu'il demande continuellement des chaînes à chiffrer jusqu'à ce que l'utilisateur ou utilisatrice entre un message vide (soit ne contenant qu'un retour à la ligne s'il ou elle appuie sur `Entrée` ou de taille zéro s'il ou elle appuie sur `Control+D`).

Conseils et remarques :

- Vous aurez peut-être besoin de sauvegarder dans un emplacement mémoire la longueur du message effectivement rentré par l'utilisateur ou utilisatrice (plutôt que dans un registre, qui peut être écrasé lors d'appels à des fonctions ou au noyau). Pour stocker un registre en mémoire vous pouvez réserver 8 octets d'un coup dans la section `bss` avec le mot clé `resq` (*reserve quadword*), par exemple : `eight_bytes: resq 1`.
- L'appel système `read` fournit des fonctionnalités de très bas niveau, et n'est normalement pas utilisé directement, en particulier en C. Un programme similaire en C utiliserait plutôt la fonction standard `fgets` et sa variante `gets`, plus robustes et plus portables.

- L'appel `read` (comme toutes les fonctions d'entrée/sortie) peur renvoyer une erreur (sa sortie est alors négative). Dans un programme destiné à être utilisé en production il faudrait vérifier ces codes d'erreur, et y répondre de façon appropriée (au minimum en imprimant un message d'erreur sur `stderr`).

2.4 Partie VII : Lecture de la clé (`step_7_message_and_key_from_stdin.asm`)

Modifier maintenant le code de la partie VI pour lire aussi la clé sur l'entrée standard. Vous devez continuer à utiliser la fichier objet `step_5_cipher_function_only.o` de la partie V sans le recompiler. Une exécution de votre programme devrait ressembler à la trace suivante :

```
$ ./step_7_executable
Enter the string to be encrypted/decrypted:
BONJOUR
Enter the key:
HOP
The encrypted/decrypted message is:
GACYAVQ
Enter the string to be encrypted/decrypted:

Goodbye!
$
```

2.5 Partie VIII : Compilation automatisée (`Makefile`)

Le but de cette dernière partie est d'utiliser la commande `make` pour compiler le programme de la partie VII, et ne recompiler que les fichiers qui en ont besoin.

2.5.1 Préliminaires

Renseignez–vous sur `make` en consultant les planches de la section *Bonus: Build Automation and Make* sur le site du cours.

Par défaut `make` lit le fichier `Makefile` du répertoire courant, et chercher à construire (*build*) la première cible rencontrée. Un fichier `Makefile` est un fichier texte qui décrit un certain nombre de règles pour construire des cibles (*target rules*). Une règle de cible est définie de la manière suivante:

```
target:  dependency1 dependency2 ...
→op(s) to create taget from dependencies
```

où → représente un caractère de tabulation (code ASCII 9). (**Attention à ne pas remplacer cette tabulation par des espaces.**)

Par exemple les lignes suivantes décrivent comment construire la cible `myProg.o` (un fichier objet) à partir du fichier source `myProg.c`. Si la cible `myProg.o` est requise, `make` n'exécutera les lignes de la règle (`echo` etc.) que si `myProg.o` n'existe pas, ou si le fichier `myProg.c` est plus récent que `myProg.o` (signifiant que `myProg.o` n'est pas à jour).

```
myProg.o:  myProg.c
→echo "Compiling myProg.o"
→gcc -c myProg.c
```

Une cible peut elle-même devenir la dépendance d'une autre cible, créant ainsi un graphe de dépendance. Par exemple dans l'exemple qui suit, la cible `myProg` dépend de `myProg.o` et `deco.o`, qui à leur tour dépendent de `myProg.c` et `deco.c` respectivement. (Les tab ne sont plus indiqués explicitement.)

```
myProg: myProg.o deco.o
  echo "Compiling myProg"
  gcc myProg.o deco.o -o myProg

myProg.o: myProg.c
  echo "Compiling myProg.o"
  gcc -c myProg.c

deco.o: deco.c
  echo "Compiling deco.o"
  gcc -c deco.c
```

2.5.2 Production d'un Makefile pour `step_7_executable`

Écrivez un makefile pour construire l'exécutable de l'étape 7. Votre makefile devra contenir au moins trois cibles : une pour l'exécutable (la première), une pour `step_7_message_and_key_from_stdin.o`, et une pour `step_5_cipher_function_only.o`. Ajoutez une dernière cible `clean` qui supprime tous les produits de compilation (les fichier `*.o` et l'exécutable). Faîte très attention pour `clean`, car vous allez utiliser la commande `rm`.

Réinitialisez la compilation avec `make clean`, puis lancez `make` deux fois. La second exécution doit afficher un message du type :

```
$ make
make: 'step_7_executable' is up to date.
```

2.5.3 Modification de `step_5_cipher_function_only.asm` et recompilation partielle

Ajouter maintenant l'impression d'un message (par exemple, "Appel de la fonction de chiffrement") dans la fonction de chiffrement contenue dans le fichier `step_5_cipher_function_only.asm`.

Sans faire `make clean`, appelez `make` pour recompiler `step_7_executable`. Qu'observez–vous ? Pourquoi ?

2.5.4 Conseils et remarques :

- Comme pour les exercices précédents, procédez par étapes, en codant chaque règle l'une après l'autre, et en vérifiant qu'elle s'exécute comme attendu. (Vous pouvez indiquer à `make` une cible spécifique à construire, par exemple `make step_7_message_and_key_from_stdin.o`.)
- Ce petit exercice ne fait qu'effleurer les capacités de `make`. Si vous souhaitez les approfondir un peu plus vous pouvez essayer les améliorations suivantes :
  - utilisez des variables dans votre Makefile (par exemple pour pouvoir changer facilement les options de `nasm`) ;
  - utilisez des variables automatiques comme `$^` ou `$@` pour rendre vos règles plus compactes (et plus faciles à maintenir) ;
  - utilisez des règles à motif (*pattern rules*), pour ne définir qu'une seule règle applicable à tous les fichiers objet (plutôt que deux règles ici).

3 Soumission et barème

3.1 Soumission

- Ajoutez les parties bonus que vous avez réalisées à votre soumission du TP 6-7-8. (Voir le TP 6-7-8 pour les détails.)

3.2 Barème

- Le TP bonus est noté sur 20, chaque partie valant 4 points.
- Divisée par 10, la note du TP bonus fournira un bonus de maximum 2 points qui seront ajoutés à votre note du TP 6-7-8 (cette dernière note restant plafonnée à 20).
- Comme précédemment, votre code sera noté en fonction de (i) sa correction, et (ii) de sa qualité (intelligibilité et commentaires).

4 Collusion, plagiat et fraude

Le code soumis doit être le résultat du travail personnel de votre binôme. L'entraide entre binômes lors d'un travail noté n'est pas interdite, mais elle doit se limiter à des conseils ponctuels et/ou de haut niveau.

Il est en revanche **strictement interdit** de soumettre tout ou partie de la solution d'un autre binôme (plagiat/copie), ou de soumettre une solution commune à plusieurs binômes, que les parties communes soient partielles ou totales (collusion). Le fait d'avoir compris le code que vous soumettez sans en être l'auteur n'est pas acceptable, car c'est ici votre capacité à produire une solution par vous-même qui est évaluée.

Des contrôles de plagiat et collusion seront réalisés sur les codes soumis. Tout étudiant ou étudiante suspecté(e) de fraude s'expose à être convoqué(e) devant la **commission disciplinaire** de l'université, et encourt des peines qui peuvent entre autres aller jusqu'à **l'exclusion temporaire ou définitive** de l'établissement.

— Fin du sujet de TP —