TP - Révisions C et assembleur

Objectifs

- Se (re)mettre à niveau sur la partie programmation bas niveau (C, assembleur)
- Comprendre le fonctionnement d'un langage bas niveau ainsi que la manipulation de registres

---

Prérequis

- Une machine Linux sous architecture x86_64 (pas ARM). Vous pouvez utiliser une VM si besoin
- Compilateur (GCC) et débugger (GDB) installés avec certains plugins (notamment `gef`).

Pour l'installation de `gcc` si vous ne l'avez pas encore sur votre machine :

```
sudo apt update
sudo apt install build-essential

# Vérifiez la version de gcc
gcc --version
```

`gdb` est par défaut installé sur la plupart des distributions mais si jamais vous avez besoin de le faire :

```
sudo apt update
sudo apt install gdb
```

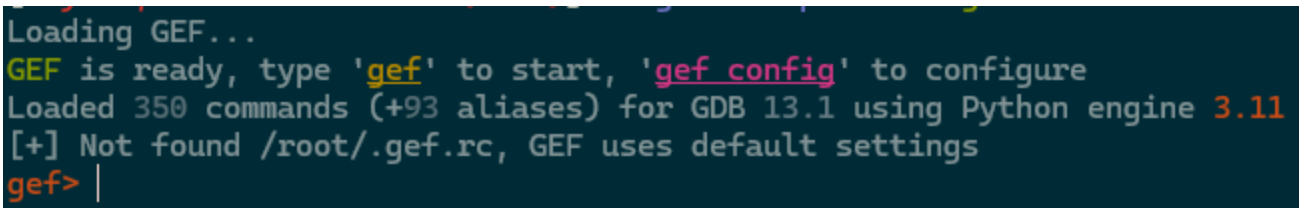
Installer le plugin `gef` pour `gdb` qui vous aidera grandement lors de vos phases de debugging :

```
wget -q https://raw.githubusercontent.com/bata24/gef/dev/install.sh -O- | sed -e 's/pip3 install/pip3
install --break-system-packages/g' | sh
```

Github associé : https://github.com/bata24/gef

Si `gef` est bien installé, vous devriez obtenir quelque chose qui ressemble à ça lors que vous ouvrirez `gdb` :

```
gdb
```



Rapport

Le TP devra être rendu en format `PDF` de préférence (Word ou Markdown également possible) à l'adresse mail : quentin.simier@synoslabs.com

Le rapport doit contenir l'ensemble des exercices avec les explications associées et les réponses demandées.
Vous devez également fournir le code produit pour ces exercices (quand cela est possible). Vous pouvez le faire via un fichier ZIP ou directement sur un Github public avec partage du lien pour que je puisse y accéder.

Vous pouvez faire le TP par groupe de 2 max. Il est important que chacun pratique et comprenne vraiment le fonctionnement des différents aspects des programmes.

N'hésitez pas à intégrer des captures d'écran dans votre rapport lorsque vous effectuez vos tests sur `gdb` par exemple pour montrer les valeurs des registres, de la stack, etc...

Énoncé

Ce TP se découpe en plusieurs parties qui suivent l'avancée du cours :

- Programmation en C
- Programmation en assembleur x86_64

ATTENTION !
Vous avez le droit d'utiliser des LLM pour vous aider dans le TP mais n'en abusez pas. Le but est de monter en compétence pour s'améliorer et être en capaciter de comprendre ce qu'on fera dans la suite des cours/TPs et non pas d'apprendre à rentrer le bon prompt.

Partie programmation en C

Pour vous remettre dans le bain et pratiquer un peu, je vous demande d'écrire un petit programme qui effectuer plusieurs opérations. Certaines choses vous sont laissées libres tandis que d'autres vous sont imposées.
Évidemment il n'existe pas une seule solution à cet exercice mais le but est de pratiquer.

1. L'objectif ici est donc de créer un petit programme qui gère des utilisateurs. On ne va pas aller jusqu'à manipuler des fichiers mais on va travailler avec les éléments suivants :

- Structures
- Tableaux
- Pointeurs
- Fonctions

Voici les éléments que votre programme doit ABSOLUMENT comporter :

- Une structure représentant un `utilisateur`. Elle doit contenir au moins 3 champs avec des types différents :
  - Un `int` pour l'ID de l'utilisateur, un tableau de `char` pour son pseudo (d'une taille de 30 caractères max. Cette taille est définie grâce à une constante de préprocesseur tout au début du programme), un `bool` pour savoir s'il est admin ou non.
- Une fonction qui va créer un nouvel utilisateur (quand on lance le programme au début, aucun utilisateur n'existe). On veut que cette fonction remplisse donc tous les champs définis par l'utilisateur (ID, pseudo et admin). Petite condition, on veut passer une structure au niveau des paramètres de la fonction.
- Une fonction d'affichage qui prend en paramètre un pointeur vers une structure et affiche tous les champs de celle-ci.
- Une fonction d'affichage global qui prend le tableau des utilisateurs et les affiche tous (uniquement leur pseudo).
- Fonction de recherche d'un utilisateur spécifique en précisant un ID. Cette fonction retourne soit la position dans le tableau, soit -1 si on ne le trouve pas.
- La liste des utilisateurs créés (dans le `main`) doivent être stockés dans un tableau.
- Une fonction qui modifie un utilisateur : on attend en paramètre le tableau des utilisateurs, un ID à rechercher dans ce tableau, un entier pour savoir quel champ modifier (0 = id, 1 = pseudo, 2 = admin. Vous pouvez utiliser le type `enum` si vous

souhaitez) ainsi que la nouvelle valeur que doit prendre ce champ. Cette fonction retourne alors le nouveau tableau des utilisateurs qui a été mis à jour.

Vous n'êtes pas obligé de créer un menu interactif dans le programme, vous pouvez indiquer les valeurs directement dans le code lors de la création d'un objet par exemple.

N'hésitez pas à faire des tests pour voir comment réagit votre programme en fonction des entrées que vous spécifiez.

Partie programmation en assembleur

Comme indiqué au début, on va écrire un petit programme en assembleur x86_64 en utilisant la syntaxe Intel.

Vous devez donc écrire un programme en assembleur qui possède un main et qui appelle une fonction avec 3 arguments. Cette fonction multiplie ces 3 éléments entre eux puis retourne le résultat.
Pas la peine de demander manuellement à l'utilisateur de rentrer des données, vous pouvez spécifier les 3 éléments en dur dans le code.
Le code de retour de ce programme doit être le résultat de la multiplication de ces trois nombres.

Par exemple, si je prends 1, 2 et 3, lorsque j'exécute mon programme, le code de retour doit être 6.
Rien n'est affiché directement à l'écran lors de l'exécution puisqu'on n'a fait aucun print dans notre programme.
Sous Linux, pour afficher le code de retour d'un programme, il faut utiliser la commande suivante juste après l'exécution du programme en question :

```
$ ./mon_programme
$ echo $?
```

On devrait alors voir s'afficher :

```
6
```

Maintenant la question est : comment faire en sorte de dire à mon programme de retourner cette valeur précisément ? Pour ce faire, on va utiliser le `syscall` exit (ressource pour vous aider : https://chromium.googlesource.com/chromiumos/docs/+/master/constants/syscalls.md#x86_64-64_bit)

Vous ferez donc deux versions de ce programme :

- La première version va placer chacun des trois arguments dans des registres différents avant l'appel de la fonction qui les multipliera. Puis à l'intérieur de la fonction on accèdera à ces arguments via les registres.
- La deuxième version va utiliser la stack pour placer les trois éléments dessus puis les récupérer une fois à l'intérieur de la fonction de multiplication. Attention, pour cette deuxième méthode, il faut qu'à la toute fin de notre programme, la stack soit vide comme au début du programme.

Le contenu de votre fichier assembleur doit commencer de cette manière pour indiquer l'utilisation de la syntaxe Intel ainsi que le début de la section `.text` qui correspond au code. La `.global` permet de spécifier l'entry point pour savoir où doit commencer la fonction main :

```
.intel_syntax noprefix

.text
.global _start

_start:
...
```

```
...
...
```

Aide

L'aide se focalise sur la compilation ainsi que des ressources pour l'utilisation de `gdb`, `objdump` et `hexedit`.

Compiler un code C :

```
gcc -o <FICHIER_EXECUTABLE> <FICHIER_C>
```

Assembler et linker un code assembleur :

```
as -o <FICHIER_OBJET_O> <FICHIER_ASM_S> && ld -o <FICHIER_EXECUTABLE> <FICHIER_OBJET_O>
```

Exemple :

```
as -o main_with_args.o main_with_args.S && ld -o main_with_args main_with_args.o
```

`gdb` cheatsheet : https://darkdust.net/files/GDB%20Cheat%20Sheet.pdf

```
gdb ./binary
```

En fonction de si vous avez du code C ou votre propre assembleur avec le label `_start` par exemple :

```
disass main
# ou
disass _start
```

Mettre un breakpoint sur main pour arrêter l'exécution à ce point :

```
b* main
```

Afficher le contenu de la stack :

```
tele
```

Aller à la prochaine instruction (next instruction). Attention ne rentre pas en détail dans les fonctions. Préférez utiliser `si` dans ce cas :

```
ni
```

Aller à la prochaine instruction (step into). Peut rentrer en détail dans les fonctions

```
si
```

Lancer un programme :

```
run
```

Afficher les infos sur les fonctions disponibles :

```
info func
```

Afficher les infos sur les registres actuels (préférer utiliser `tele` avec `gef`)

```
info register
```