TP noté : exploration de la mémoire d'un programme C

Consignes de présentation

Pour chaque question, indiquer son numéro puis, juste en dessous, copier-coller **deux lignes du terminal** :
— la commande exécutée ;
— une seule ligne de sa réponse, choisie pour justifier la réponse à la question.

La réponse doit être aussi ciblée que possible : ne conserver que la ligne pertinente pour la question posée. Privilégier les options de la commande ou un filtrage avec **grep** permettant de limiter directement la sortie aux informations demandées.

On peut consulter le mode d'emploi de **grep**, notamment les options **-B** et **-A**, qui permettent d'afficher quelques lignes respectivement avant (*before*) et après (*after*) une ligne contenant le motif recherché.

Par exemple, la commande suivante ne filtre que la ligne contenant le mot « Boutisme » dans la descripription de l'architecture du processeur et donne aussi la ligne juste au dessus et les deux lignes en dessous :

```
$ lscpu | grep -A 2 -B 1 Boutisme
Address sizes:                        39 bits physical, 48 bits virtual
Boutisme :                            Little Endian
Processeur(s) :                       8
Liste de processeur(s) en ligne :     0-7
```

Ne pas fournir de capture d'écran. Conserver les numéros des questions dans l'ordre du sujet.

Programme étudié

Créer un fichier **memoire.c** contenant le programme suivant.

```c
int globale_initialisee = 42;
int globale_non_initialisee;
const int constante = 2026;

int main(void) {
    int locale = 7;
    static int locale_statique = 13;

    return globale_initialisee + globale_non_initialisee
           + locale + locale_statique;
}
```

Dans ce TP, on utilise une machine Linux avec un processeur x86-64. Les commandes sont à exécuter dans le répertoire contenant **memoire.c**.

1 Exploration du fichier objet

Compiler le programme sans effectuer l'édition de liens :

```
gcc -Wall -Wextra -O0 -g -fno-omit-frame-pointer -mno-red-zone -c memoire.c -o
    memoire.o
```

L'option **-c** produit un *fichier objet*. Celui-ci contient notamment du code machine, des données et une table des symboles.

Les options utilisées désactivent les optimisations et facilitent l'observation des trames de pile dans la seconde partie du TP.

Utilisation de objdump

La *table des symboles* répertorie notamment les noms des fonctions et des variables globales ou statiques. Elle indique leur section, leur position dans cette section et leur taille. La commande suivante affiche la table des symboles :

```
objdump -t memoire.o
```

Pour les variables et les fonctions définies dans ce fichier objet, chaque ligne se lit de gauche à droite :

1. le déplacement du symbole depuis le début de sa section, en octets et en hexadécimal ;
2. sa liaison : **g** pour un symbole global, accessible depuis d'autres fichiers objets, ou **l** pour un symbole local au fichier objet ;
3. sa nature : **O** pour un objet de données, ou **F** pour une fonction ;
4. le nom de sa section ;
5. sa taille en octets, écrite en hexadécimal ;
6. son nom.

Par exemple, une ligne fictive pourrait être :

```
0000000000000008 l     O .data  0000000000000004 compteur.0
```

Elle décrit un objet de données nommé **compteur.0**, local au fichier objet, occupant quatre octets dans **.data** à partir du déplacement 8.

Attention : l ne signifie pas nécessairement « variable locale à une fonction ». Une variable déclarée `static` au niveau global possède également une liaison locale au fichier objet.

Un déplacement nul signifie que l'objet commence au début de sa section ; ce n'est pas son adresse virtuelle lors de l'exécution. Plusieurs symboles peuvent désigner une même position : on ne peut pas en déduire que l'objet est le seul symbole placé à cet endroit.

Les *sections* du fichier organisent notamment le code et les données destinés à être placés en mémoire. Plusieurs sections peuvent être regroupées dans un même segment chargé en mémoire.

La commande suivante affiche les octets enregistrés dans les sections du fichier :

```
objdump -s memoire.o
```

Pour limiter cet affichage à une section, on ajoute **-j** suivi de son nom. Par exemple :

```
objdump -s -j .data memoire.o # affichage de la section .data
```

Dans chaque ligne de contenu, on trouve une position en hexadécimal, les octets stockés regroupés par paquets de quatre, puis une représentation de ces octets sous forme de caractères. Les points de la dernière colonne représentent les caractères non affichables ; cette colonne ne donne pas les valeurs des entiers.

Table des symboles

Pour chaque question, isoler la ligne du symbole demandé avec **objdump -t** et un filtrage adapté. Lire les différentes colonnes pour vérifier toutes les propriétés indiquées dans la question. Le rendu reste constitué de la commande et d'une seule ligne pertinente de sa réponse.

Le compilateur peut ajouter un suffixe au nom de la variable locale statique.

Question 1.

Afficher la ligne de `globale_initialisee`. Identifier sa section et son déplacement dans cette section. Repérer les indications montrant qu'il s'agit d'un objet de données à liaison globale, de taille quatre octets.

Question 2.

Afficher la ligne de `globale_non_initialisee`. Identifier sa section, son déplacement et sa taille. Vérifier qu'il s'agit également d'un objet à liaison globale, mais que son déplacement est mesuré depuis le début d'une autre section que celle de `globale_initialisee`.

Question 3.

Afficher la ligne de `locale_statique`. Identifier sa section, son déplacement et sa taille. Repérer l'indication de liaison qui la distingue des deux variables globales précédentes. Une variable locale statique possède ainsi un emplacement dans une section de données, malgré sa déclaration dans une fonction.

Question 4.

Afficher la ligne correspondant à `main`. Identifier sa section, son déplacement et sa taille en hexadécimal. Repérer l'indication qui distingue une fonction d'un objet de données. La taille affichée correspond au nombre d'octets de son code machine, pas à la taille de sa trame de pile.

Question 5.

Afficher la ligne de la table des symboles correspondant à `constante`. Identifier sa section, son déplacement et sa taille.

Question 6.

Vérifier que la variable `locale` est absente de la table des symboles. Expliquer pourquoi. Pour cette question, fournir la commande et une courte explication.

Question 7.

En utilisant les options de **grep**, afficher en une seule commande les informations sur `main`, la constante et les variables globales.

Contenu des sections

Sur la machine utilisée, un `int` occupe quatre octets. Les entiers sont représentés en *petit-boutiste* : l'octet de poids faible est stocké à l'adresse la plus petite.

Ainsi, l'entier 1 est représenté par les quatre octets **01 00 00 00**, affichés **01000000** par **objdump -s**.

Question 8.

Afficher le contenu de la seule section contenant `globale_initialisee`. À partir du déplacement et de la taille lus dans la table des symboles, repérer les quatre octets de cette variable et vérifier qu'ils représentent 42, soit **2a** en hexadécimal. Fournir la commande et la ligne contenant ces octets, puis indiquer les quatre octets concernés et la valeur entière qu'ils représentent.

Question 9.

À partir du déplacement et de la taille de `locale_statique`, afficher uniquement les octets correspondant à cette variable en utilisant les options −−start−address et −−stop−address. Vérifier que ces octets représentent l'entier 13 (**0d**). Fournir la commande et la ligne contenant ces octets.

La section **.bss** réserve de la mémoire initialisée à zéro lors du chargement. Ses octets initiaux ne sont pas enregistrés dans le fichier objet : **objdump -s** n'en affiche donc pas le contenu. La table des symboles décrit néanmoins les objets qui y sont placés.

La commande suivante affiche les caractéristiques des sections :

```
objdump -h memoire.o
```

La colonne **Size** donne leur taille en hexadécimal.

Question 10.

Isoler la ligne donnant le nom et la taille de la section **.bss**. Vérifier que cette taille permet de réserver les quatre octets de `globale_non_initialisee`.

2 Observation de la pile avec gdb

Produire maintenant un fichier exécutable :

```
gcc -Wall -Wextra -O0 -g -fno-omit-frame-pointer -mno-red-zone -no-pie memoire.c -o
    memoire
```

Puis lancer le débogueur :

```
gdb ./memoire
```

Quelques commandes

Dans **gdb**, saisir les commandes suivantes pour préparer l'affichage :

```
set pagination off
set disassembly-flavor intel
```

On utilisera les commandes suivantes :

disassemble main Affiche les instructions machine de **main**.

break *main Place un point d'arrêt sur la toute première instruction machine de **main**.

run Lance le programme jusqu'au prochain point d'arrêt.

stepi Exécute une instruction machine.

x/i $pc Affiche la prochaine instruction machine à exécuter.

p/x $rsp Affiche le contenu de **RSP** en hexadécimal.

p/x $rbp Affiche le contenu de **RBP** en hexadécimal.

Attention : lorsque **gdb** affiche une instruction comme instruction courante, celle-ci **n'a pas encore été exécutée**.

Entrée dans main

Saisir :

```
break *main
run
disassemble main
```

Repérer le prologue de **main**, contenant notamment :

```
push rbp
mov rbp, rsp
```

D'autres instructions peuvent précéder ce prologue.
Avancer avec **stepi** jusqu'à ce que la prochaine instruction à exécuter soit **push rbp**. On peut à tout moment vérifier sa position ou la prochaine instruction à réaliser avec **x/i $pc**.

Question 11.

Afficher **RSP** et copier la commande et la ligne obtenue.
Mémoriser cette valeur dans une variable du débogueur :

```
set $rsp_avant_push = $rsp
```

Cette commande crée une variable de **gdb** ; elle ne modifie pas le programme étudié.
Exécuter l'instruction **push rbp** avec **stepi**.

Question 12.

Afficher **RSP** et copier la ligne obtenue. L'adresse a-t-elle baissé ou augmenté ?

Question 13.

Afficher la variation de **RSP** avec :

```
p/d (long)$rsp - (long)$rsp_avant_push
```

Copier la ligne obtenue.

Installation du pointeur de trame

Question 14.

Afficher le registre **RBP**.
La prochaine instruction doit être maintenant :

```
mov rbp, rsp
```

Question 15.

Vérifier que c'est bien le cas.

Question 16.

L'exécuter puis afficher l'instruction suivante. Afficher ensuite **RBP** et copier les lignes obtenues.

Question 17.

Vérifier que **RBP** et **RSP** contiennent la même adresse en saisissant :

```
p $rbp == $rsp
```

Copier la ligne obtenue.

Réservation de l'espace pour les variables locales

Repérer dans le listing du désassemblage l'instruction qui réserve de l'espace dans la pile en soustrayant une quantité à **RSP**. Avancer si nécessaire jusqu'à ce que la prochaine instruction à réaliser soit cette réservation.

Question 18.

Copier la ligne du désassemblage contenant cette instruction.
Avancer si nécessaire jusqu'à cette instruction, sans encore l'exécuter, puis mémoriser les deux registres :

```
set $rsp_avant_reservation = $rsp
set $rbp_avant_reservation = $rbp
```

Exécuter l'instruction de réservation.

Question 19.

Afficher le nombre d'octets réservés avec :

```
p/d (long)$rsp_avant_reservation - (long)$rsp
```

Copier la ligne obtenue.

Question 20.

Vérifier que **RBP** n'a pas changé en saisissant :

```
p $rbp == $rbp_avant_reservation
```

Copier la ligne obtenue.

Adresse de la variable locale

Continuer avec **stepi** et s'arrêter juste avant l'instruction qui place la valeur 7 dans la variable `locale`.

Question 21.

Afficher cette instruction puis l'exécuter. Pour cette question, copier les deux commandes et leurs réponses.

Question 22.

Cette écriture a-t-elle modifié le pointeur de pile ? Justifier la réponse et fournir une commande permettant de le vérifier, accompagnée de son résultat.

Question 23.

Afficher la valeur de `locale` avec **p locale** et copier la ligne obtenue.

Question 24.

Afficher son adresse avec **p/x &locale** et copier la ligne obtenue.

Question 25.

Afficher le déplacement de cette variable par rapport au pointeur de trame :

```
p/d (long)&locale - (long)$rbp
```

Copier la ligne obtenue.

L'addition

Question 26.

Repérer dans le désassemblage où se situent les instructions qui effectuent l'addition des 4 termes. Expliquer quels registres sont impliqués et décrire en Français le déroulement de la somme.

Pour cette question la consigne change un peu : copier toutes les instructions concernées et ajouter lexplication demandée.

EAX est le registre utilisé pour transmettre une valeur de retour de type `int`. Passer toutes les exécutions des instructions d'addition et la dernière modification de **EAX**.

Continuer avec **stepi** jusqu'à avoir exécuté la dernière addition, sans exécuter l'instruction **leave**.

Question 27.

Afficher en décimal le contenu du registre utilisé pour transmettre la valeur de retour. Vérifier que le résultat est bien égal à 62.

Libération de la trame

La prochaine instruction à exécuter est **leave**. L'exécuter avec **stepi**.

Cette instruction libère l'espace réservé aux variables locales et restaure l'ancien **RBP**. Le registre **RSP** pointe alors sur l'adresse de retour.

Question 28.

Vérifier avec **gdb** que **RSP** a retrouvé sa valeur précédant **push rbp**. Copier la commande et la ligne obtenue.

La prochaine instruction est **ret**. Elle dépile l'adresse de retour et la place dans le pointeur d'instruction **RIP** : l'exécution reprend ainsi dans le code appelant. Le registre **RSP** augmente de 8 octets.

Dans le cas de `main`, le code appelant appartient à l'environnement d'exécution C. Il termine ensuite le programme en utilisant la valeur retournée comme statut de sortie.

Question 29.

Mémoriser la valeur courante de **RSP** dans une variable au nom bien choisi.

Exécuter **ret** avec **stepi**, puis vérifier que **RSP** a augmenté de 8 octets. Copier la commande de vérification et la ligne obtenue.

Quitter le débogueur avec **quit**.