TD/TP4 - Examen de la mémoire

Traitez les questions dans l'ordre. Ne cherchez pas à brûler les étapes. Avancez pas à pas !
Connectez-vous sur la machine virtuelle Linux / CentOS (id : iut, mdp : iut).
À la fin de chaque séance, pensez à sauvegarder votre travail si nécessaire.
Les programmes sont disponibles sur https://gitlab.com/Thomas-Hugel/IUT/PPN-2013/M2101-Bas-Niveau

1 Premiers pas

1.1 Observation de l'exécutable

1. Créez un répertoire `M2101/HelloWorld`. Mettez-y le programme appelé `HelloWorld.c`. Compilez-le avec l'option de débogage dans un fichier `HelloWorld.out`. Comparez la taille de ces deux fichiers. Ouvrez le fichier `HelloWorld.out` dans un éditeur de texte (`gedit`), et cherchez-y la chaîne «main».
2. Afin de rendre le code assembleur lisible par un humain, tapez la commande suivante :

```
objdump -M intel -Dtx HelloWorld.out > HelloWorld.out.txt
```

3. Ouvrez le fichier obtenu, et cherchez-y cette fois le code de la fonction `main`.
4. À l'aide de la commande `man ascii`, déterminez l'encodage ASCII hexadécimal de "Bonjour Limoges !\n".
5. Regardez dans l'assembleur comment cette chaîne est traitée. Remarquez bien l'ordre d'affichage : l'affichage est tel qu'il permet de lire directement les adresses de 64 bits, et l'architecture est petit-boutiste...

Listing 1 – HelloWorld.c

```c
#include <stdio.h>

int main(void) {
        char message[] = "Bonjour Limoges !\n";
        for (int i = 0; i < 10; i++)
                printf("%s\n", message);
        return 0;
}
```

```
00000000004004f6 <main>:
  4004f6:       55                      push   rbp
  4004f7:       48 89 e5                mov    rbp,rsp
  4004fa:       48 83 ec 20             sub    rsp,0x20
  4004fe:       48 b8 42 6f 6e 6a 6f    movabs rax,0x2072756f6a6e6f42
  400505:       75 72 20
  400508:       48 89 45 e0             mov    QWORD PTR [rbp-0x20],rax
  40050c:       48 b8 4c 69 6d 6f 67    movabs rax,0x207365676f6d694c
  400513:       65 73 20
  400516:       48 89 45 e8             mov    QWORD PTR [rbp-0x18],rax
  40051a:       66 c7 45 f0 21 0a       mov    WORD PTR [rbp-0x10],0xa21
  400520:       c6 45 f2 00             mov    BYTE PTR [rbp-0xe],0x0
  400524:       c7 45 fc 00 00 00 00    mov    DWORD PTR [rbp-0x4],0x0
  40052b:       eb 10                   jmp    40053d <main+0x47>
  40052d:       48 8d 45 e0             lea    rax,[rbp-0x20]
  400531:       48 89 c7                mov    rdi,rax
  400534:       e8 b7 fe ff ff          call   4003f0 <puts@plt>
  400539:       83 45 fc 01             add    DWORD PTR [rbp-0x4],0x1
  40053d:       83 7d fc 09             cmp    DWORD PTR [rbp-0x4],0x9
  400541:       7e ea                   jle    40052d <main+0x37>
  400543:       b8 00 00 00 00          mov    eax,0x0
  400548:       c9                      leave
  400549:       c3                      ret
  40054a:       66 0f 1f 44 00 00       nop    WORD PTR [rax+rax*1+0x0]
```

1.2 Utilisation du débogueur gdb

1. En une ligne de commande, ajoutez «`set disassembly intel`» à la fin du fichier `~/.gdbinit` (c'est pour configurer l'affichage dans `gdb`).
2. Lancez les commandes suivantes et analysez ce que vous voyez (faites le lien avec le code source) :

```
gdb ./HelloWorld.out
list
b main
r
disass main
n
disass main
n 10
disass main
```

```
x/1wd &i
x/1wd $rbp-4
ni
x/1wd &i
ni
x/5i $rip
ni
disass main
x/12gx $rsp
x/12gx $rbp
```

3. Quelle est l'adresse de `i` ?
4. Faites un dessin de la situation, en faisant apparaître `main`, `i`, `rip`, `rsp` et `rbp`.
5. Tapez `q` pour sortir du débogueur.

2 Appel d'une fonction

1. Récupérez le fichier `FunctionCall.c` et complétez le tableau de droite à l'aide de gdb.
2. Pour le dessin qui est demandé, faites un `x/10gx $rsp` pour vous aider.
3. La fonction `to_seconds()` travaille-t-elle directement sur `hh`, `mm`, `ss` et `total` ?
4. Comment les paramètres sont-ils passés à la fonction `to_seconds()` ?
5. Comment la valeur retournée est-elle transmise à `main()` ?
6. Relancez le débogage et déterminez l'adresse de retour de `to_seconds()` et de `main()`.
7. À quoi correspond l'adresse qui est empilée juste au-dessus de l'adresse de retour ?

Listing 2 – FunctionCall.c

```
#include <stdio.h>

int to_seconds (int hours, int minutes, int seconds) {
        int result = hours + 60 * minutes + 3600 *
             seconds;
        hours = minutes = seconds = 0;
        printf ("Cela fait %d secondes.\n", result);
        return result;
}

int main(void) {
        int hh = 10;
        int mm = 33;
        int ss = 28;
        int total = to_seconds (hh, mm, ss);
        return 0;
}
```

arrêt	*var.*	*adresse*	*valeur*
l. 14	`main`		
	`to_seconds`		
	`rip`	-	
	`rsp`	-	
	`rbp`	-	
	`hh`		
	`mm`		
	`ss`		
	`total`		
l. 4	`edi`	-	
	`esi`	-	
	`edx`	-	
	`rip`	-	
	`rsp`	-	
	`rbp`	-	
	-	backtrace	
Faites un dessin de l'état de la pile, avec les cadres des fonctions.			
l. 15	`total`		
	`eax`	-	
	`rip`	-	
	`rsp`	-	
	`rbp`	-	
	`hh`		
	-	backtrace	

3 Les segments de la mémoire

1. Récupérez le fichier `MemorySegments.c`.
2. À l'aide du débogueur, complétez le tableau de droite.
3. Faites un dessin où vous placerez `a`, `b`, `c`, `counter`, `LEN`, `ptr_a`, `ptr_LEN`, `tab`.
4. Expliquez la différence entre `counter` et `c`.
5. Que vaut `(tab + 1) - tab` ?
6. Expliquez la différence entre `free(tab)` et `tab=NULL`.
7. Pourquoi est-il conseillé de faire `tab=NULL` après `free(tab)` ?

arrêt	*var.*	*adresse*	*valeur*	*segment*
l. 10	`counter`			
	`c`			
l. 17	`total`			
l. 10	`counter`			
	`c`			
l. 24	`a`			
	`ptr_a`			
	`*ptr_a`			
	`LENGTH`			
	`ptr_LEN`			
	`tab`			
	`*tab`			
	`tab[0]`			
	`tab[1]`			
	`*(tab + 1)`			
l. 25	`tab`			
	`*tab`			
l. 27	`tab`			
	`*tab`			

Listing 3 – MemorySegments.c

```
1  #include <stdlib.h>
2
3  const int LENGTH = 10;
4
5  int to_minutes (int hours, int minutes) {
6          static int counter = 0;
7          ++counter;
8          int c = 0;
9          ++c;
10         return hours + 60 * minutes;
11 }
12
13 int main(void) {
14         int a = 18;
15         int b = 33;
16         int total = to_minutes (a, b);
17         total = to_minutes (b, a);
18
19         int *ptr_a = &a;
20         const int *ptr_LEN = &LENGTH;
21         int *tab = malloc (LENGTH * sizeof(int));
22         tab[0] = 31;
23         *(tab + 1) = 43;
24         free (tab);
25         tab = NULL;
26
27         return 0;
28 }
```

4 Pointeurs

4.1 Un peu d'arithmétique

Dans un fichier `Sum.c`, écrivez deux implémentations de la fonction `double sum (double *tab, int size)` qui calcule la somme des éléments d'un tableau :

1. une version `sum1` qui utilise un indice pour se déplacer dans le tableau ;
2. une version `sum2` qui n'utilise pas d'indice auxiliaire (mais l'arithmétique des pointeurs).

Testez votre code.

4.2 Passage par adresse

Dans un fichier `Swap.c` écrivez une fonction `void swap (double *x, double *y)` qui échange les valeurs de `x` et `y`. Testez votre code. Que se passerait-il sans les pointeurs ?

4.3 Segments de la mémoire

1. Récupérez le fichier `Pointers.c` et lisez le code source.
2. Essayez de deviner ce qui va se passer et s'afficher.
3. Compilez-le, et exécutez-le pour vérifier.
4. Quels sont les défauts de ce programme ? Comment les corriger ?

Listing 4 – Pointers.c

```c
#include <stdio.h>
#include <stdlib.h>

int* f() {
        static int a = 0;
        ++a;
        return &a;
}

int* g() {
        int *b = malloc (sizeof(int));
        *b = 0;
        ++(*b);
        return b;
}

int* h() {
        int c = 0;
        ++c;
        return &c;
}

int main (void) {
        int *result_f, *result_g, *result_h;
        result_f = f();
        result_g = g();
        result_h = h();
        result_f = f();
        result_g = g();
        result_h = h();
        printf("*result_f = %d\n", *result_f);
        printf("*result_g = %d\n", *result_g);
        printf("*result_h = %d\n", *result_h);
        return 0;
}
```

Listing 5 – Strings.c

```c
#include <stdio.h>
#include <string.h>

int main() {
        char hello[] = "Bonjour";
        char letters[16] = "abcdefghijklmnop";
        char numbers[15] = "012345678901234";
        char bye[11] = "Au revoir";
        int x = 15, y = 31;

        printf("hello: %s\n", hello);
        printf("letters: %s\n", letters);
        printf("numbers: %s\n", numbers);
        printf("numbers + 7: %s\n", numbers + 7);
        printf("bye: %s\n", bye);
        return 0;
}
```

5 Chaînes de caractères

5.1 Concaténation

Dans un fichier `StringConcatenation.c`, écrivez deux implémentations de `char *strcat(char *dest, const char *src)` d'après la description fournie par `man strcat` :

1. une version `strcat1` qui utilise des indices pour se déplacer dans les chaînes de caractères ;
2. une version `strcat2` qui n'utilise pas d'indices auxiliaires (mais l'arithmétique des pointeurs).

Écrivez les préconditions dans un cartouche, et testez votre code.

5.2 Petit problème...

1. Récupérez le fichier `Strings.c`, lisez-le et essayez de deviner ce qui va s'afficher.
2. Compilez-le (en mode débogage) et exécutez-le. Quel est le problème ?
3. Avec gdb, arrêtez-vous à la ligne 11, et examinez 12 mots géants en hexadécimal à partir du sommet de la pile.
4. Retrouvez les valeurs du code source.
5. Expliquez ce qui se passe, et en particulier pourquoi il y a un problème avec `letters` mais pas avec `numbers` (indication : tenez compte de ce qu'on appelle *l'alignement* des valeurs sur la pile).
6. Récrivez ce programme en faisant les corrections nécessaires.
7. Un tel problème peut-il se poser en Java ? pourquoi ?