Université d'AIN TEMOUCHENT – Algérie

Faculté des Sciences et de la Technologie

Département de Mathématiques et d'Informatique

Matière: **Système d'exploitation avancé**

Chapitre 05 : Gestion de la pile et du Tas	**TD/TP 5- Segmentation de la mémoire en C**

Exercice 01: *//Analyse des sections avec size*

1) Soit le programme suivant nommé `prog1.c`, examinons la taille de ses différentes sections.

```
#include <stdio.h>
int main(void) {
return 0;
}
```

Compilez puis évaluez la taille de ce programme à l'aide de la commande `size`.

```
$ gcc prog1.c -o prog1
$ size prog1
```

Résultat :

```
text        data    bss     dec     hex     filename
1440        292     4       1736    6c8     prog1
```

2) Expliquez la signification des différentes sections et leurs fonctions.

3) Que représentent les champs `dec` et `hex` ?

4) Ajoutez une variable globale non initialisée de type entier nommée `globale` et étudiez les tailles des différentes sections à nouveau. Quelle taille de section a été modifié et pourquoi ?

5) Ajoutez maintenant une variable statique de type entier nommé `var` à l'intérieur de la fonction `main()` et étudiez les tailles des différentes sections à nouveau. Quelle taille de section a été modifié et pourquoi ?

6) Initialisez la variable `var` par 10 et étudiez les tailles des différentes sections à nouveau. Quelle taille de section a été modifié et pourquoi ?

7) Maintenant initialisez la variable globale `global` par 200 et étudiez les tailles des différentes sections à nouveau. Quelle taille de section a été modifié et pourquoi ?

Exercice 02: *// Cartographie mémoire simple*

1) Compilez puis exécutez le programme suivant nommé `prog2.c`:

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

int global_initialized = 42;
int global_uninitialized;

void function_example() {}

int main() {
    int local_var = 0;
    int *heap_var = malloc(sizeof(int));
    *heap_var = 123;

    printf("Adresse du code: %p\n", (void*)function_example);
    printf("Adresse global initialisée: %p\n", (void*)&global_initialized);
    printf("Adresse global non initialisée: %p\n",
(void*)&global_uninitialized);
    printf("Adresse variable locale: %p\n", (void*)&local_var);
```

```
    printf("Adresse mémoire dynamique: %p\n", (void*)heap_var);

    free(heap_var); // Libérer la mémoire
    return 0;
}
```

2) Que signifie les symboles suivants: `%p`, et `(void*)` ?

3) Expliquez les différentes valeurs des adresses affichées et leur relation avec les segments: *TAS, pile, data* et *text*.

4) Schématisez les différents segments en spécifiant les adresses.

5) Une autre manière de voir les segments de la mémoire est d'utiliser l'outil `pmap` avec le PID du processus en cours d'exécution.

```
pmap <PID>
```

Attention, il faut modifier le programme.

Exercice 03: *//Croissance Tas/Pile*

1) Compilez puis exécutez le programme suivant nommé prog3.c:

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

void recursive_function(int depth) {
    int stack_var; // Variable locale
    printf("Profondeur %d - Adresse stack : %p\n", depth, (void*)&stack_var);
    if (depth > 0) recursive_function(depth - 1);
}

int main() {
    int *heap_var1 = malloc(100);
    int *heap_var2 = malloc(100);

    printf("Adresse heap 1 : %p\n", (void*)heap_var1);
    printf("Adresse heap 2 : %p\n", (void*)heap_var2);

    recursive_function(3);

    free(heap_var1);
    free(heap_var2);
    return 0;
}
```

2) Schématisez les différents segments ainsi que les cadres de pile.

3) Dans quel sens croit les zones de la pile et du TAS ?

Exercice 04: *// Analyse de /proc/<PID>/maps*

1) Compilez puis exécutez le programme suivant nommé `prog4.c`:

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

/* constantes et variables globales */
const int const_int = 0;
int global_int = 0;
int global_array[] = {1, 2, 3};
char global_string[50];

int main(void)
```

```
{
  /* variables locales a la fonction main() */
  int i = 0;
  int* pti =  &i;
  char* static_local_string = "tototo";
  char* dynamic_local_string = (char*) malloc(512);
  int local_array[2];
  static int static_integer = 10;
  char c;

  strcpy(global_string, "-------------------------");

  printf("%s\n", global_string);
  printf("&const_int = %p\n&global_int = %p\nglobal_array = %p\nglobal_string =
%p\n", &const_int, &global_int, global_array, global_string);
  printf("%s\n", global_string);
  printf("&i = %p (pti = %p, &pti = %p)\n", &i, pti, &pti);
  printf("%s\n", global_string);
  printf("&static_local_string = %p (static_local_string = %p)\n",
&static_local_string, static_local_string);
  printf("&dynamic_local_string = %p (dynamic_local_string = %p)\n",
&dynamic_local_string, dynamic_local_string);
  printf("%s\n", global_string);
  printf("local_array = %p\n", local_array);
  printf("&static_integer = %p\n", &static_integer);
  printf("%s\n", global_string);
  printf("&main = %p, &strcpy = %p\n", main, strcpy);

  /* attente pour recuperation du numero de processus */
  scanf("%c", &c);
  return 0;
}
```

2) Schématisez les différents segments en spécifiant les adresses.

3) Dans quelle section est allouée l'identificateur `dynamic_local_string` ? Et la zone mémoire sur laquelle il pointe ? Justifiez votre réponse.

4) Justifiez la présence des identificateurs déclarés dans le `main()` dans la portion de la mémoire du processus correspondant à la pile.

5) Cherchez le PID de votre processus <PID>, et tapez la commande:

```
$ cat /proc/<PID>/maps
```

6) Identifiez la zone de code (text): quelles sont ses protections et pourquoi ? Faites de même avec la zone de données et répondez aux mêmes questions. Enfin, identifiez les régions en indiquant leur adresse qui correspondent à la pile et au tas du processus.

Exercice 05 : //Analyse de la pile avec GDB

1) Installez GDB si nécessaire :

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

2) Soit le programme pile.c suivant:

```
#include <stdio.h>
void fonction(int a) {
int x = 10;
int y = 20;
printf("Dans fonction(): a=%d, x=%d, y=%d\n", a, x, y);
}
```

```
int main() {
int m = 5;
fonction(m);
return 0;
}
```

Compilez sans optimisation et avec symboles de debug :

```
$ gcc -g -O0 pile.c -o pile
```

3) Lancez GDB :

```
$ gdb ./pile
```

4) Lancez le programme :

```
(gdb) run
```

Dans GDB :

```
(gdb)break fonction
(gdb)run
```

Voici ce que le débogueur affiche

```
Breakpoint 1, fonction (a=5)
```

5) Afficher la pile d'appels en tapant la commande:

```
(gdb) backtrace
```

ou :

```
(gdb) bt
```

Le débogueur affiche

```
#0 fonction
#1 main
```

6) Pour afficher les variables locales taper

```
(gdb) info locals
```

Résultat attendu :

```
x = 10
y = 20
```

7) Pour Afficher les arguments de fonction

```
(gdb) info args
```

Résultat :

```
a = 5
```

8) Pour Obtenir les adresses dans la pile

```
(gdb) print &a
(gdb) print &x
(gdb) print &y
```

Que remarquerez-vous :

9) Pour avoir un Affichage brut de la pile, Obtenez l'adresse de x :

```
(gdb) print &x
```

Puis examinez la mémoire :

```
(gdb) x/8xw &x
```

Interprétation :

x : examine

8 : nombre de mots

x : hexadécimal

w : word (4 octets)

10) Pour Voir le pointeur de pile (ESP/RSP), Taper:

```
(gdb) info registers
```

Si vous êtes sur une machine 32 bits, cherchez "esp", sinon (Sur machine 64 bits), cherchez "rsp", puis taper:

```
(gdb) x/8xw $rsp
```

Si "No registers", taper :

```
(gdb) set architecture i386
```

ou assurez-vous que le programme est lancé :

```
(gdb) run
```

11) Pour faire un suivi Pas à pas (instruction par instruction) taper:

```
(gdb) step
(gdb) next
```

Comparez :

step : entre dans la fonction

next : saute l'appel de fonction

12) Pour Changer la valeur directe dans la pile

```
(gdb) set variable x = 99
(gdb) continue
```

Le programme affichera :

```
x = 99
```

Exercice 06: //Faire Déborder Un Buffer

1) Modifiez pile.c :

```
#include <stdio.h>
void vuln() {
char buffer[8];
gets(buffer);
}
int main() {
vuln();
return 0;
}
```

2) Compile :

```
$ gcc -fno-stack-protector -z execstack -g pile.c -o pile
```

ignorer les erreurs et lancer:

3)
```
$ gdb ./pile
```

Dans GDB :

```
(gdb) run
```

4) Tapez une longue chaîne :

```
(gdb) AAAAAAAAAAAAAAAAAAAA
```

Observation :

```
Crash
overwrite de la pile
```

5) Afficher l'adresse de retour :

```
(gdb) info frame
```

Observez :

```
saved rip
```