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 5: *// Segmentation complète d'un programme statique / dynamique*

La mémoire virtuelle d'un processus est constituée d'un ensemble de régions. Une région est une portion contiguë de la mémoire virtuelle d'un processus. À chaque région le système associe des protections et un rôle particulier. Le but de cet exercice est de visualiser et de comprendre ces régions.

1) Commencez par compiler (avec l'option -static) le programme suivant :

```
#include <sys/types.h>
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>

int une_globale;
```

```
int une_autre=4;
char* alloc;

int main()
{
int une_locale;
alloc = malloc(1024L * 1024L * 10L); /* 10mo */
printf("PID = %d\n", getpid());
printf("adresse de une_globale = %8lx\n",(unsigned long)&une_globale);
printf("adresse de une_autre_globale = %8lx\n",(unsigned long)&une_autre);
printf("adresse de une_locale = %8lx\n", (unsigned long)&une_locale);
printf("adresse de alloc = %8lx\n", (unsigned long) alloc);
printf("adresse de main = %8lx\n", (unsigned long) & main);
printf("adresse de printf = %8lx\n", (unsigned long) & printf);
/* afficher la carte mémoire */
sprintf(alloc, "cat /proc/%d/maps", getpid());
system(alloc);
return 0;
}
```

2) En comparant l'espace adressable de chaque région et les adresses données par le programme, donnez un sens à chaque région.

3) Refaire le même exercice mais en compilant le programme sans la directive **-static**.
Remarque : le système UNIX associe un numéro à chaque fichier sur disque. Ce numéro (appelé le numéro de i-node) peut-être visualisé en utilisant l'option -i de la commande ls.

Exercice 6: *//Informations sur la mémoire*

1) En utilisant **ps**, afficher la liste de tous les processus avec les informations suivantes : la taille de la mémoire virtuelle du processus, la taille de la mémoire physique utilisée par le processus, et le pourcentage de mémoire physique utilisée par le processus par rapport à la mémoire physique totale du système.

2) Que fait la commande **free** ? Comment faire pour obtenir un nouvel affichage toutes les 5 secondes ?

3) Le programme **free** utilise **/proc/meminfo**. Comment utiliser ce dernier pour afficher sur une seule ligne la quantité de mémoire disponible sur le système ?

4) Quelles informations peut-t-on obtenir en utilisant la commande **vmstat** ? Noter en particulier celles relatives à la gestion de la mémoire.

5) Que fait la commande : **cat /proc/<pid>/status**

Exercice 7: *//Protection de la mémoire avec Linux*

Commencez par étudier l'appel système **mprotect** : **man mprotect**

Utilisez cette fonction pour protéger une portion mémoire d'un de vos programmes. La démarche est la suivante :

1) Écrivez un programme qui alloue un tableau de 16 kilo-octets (fonction malloc) et qui initialise ce tableau.

2) Modifiez votre programme pour protéger en écriture (puis en lecture) une portion centrale de votre tableau. Quel est le résultat ? Tentez sur votre tableau une opération interdite.
Remarques:
- *Vous aurez surement besoin d'aligner un pointeur sur le début d'une page : utiliser la fonction **sysconf** pour récupérer la taille d'une page.*
- *Vous pouvez également utiliser la fonction **memalign**.*

3) Affichez la carte mémoire avant la protection et après. Quel est l'effet de la protection sur les régions du processus ?