ASR TD2
Du programme au processus

L'organisation mémoire d'un programme est définie par le développeur suivant la manière dont il déclare ses variables. Cette organisation a des conséquences sur le comportement du programme une fois chargé par le système d'exploitation, notamment sur sa sécurité et sa performance. Il est donc important de bien comprendre les implications liées aux différentes manière de déclarer des variables.

Organisation mémoire d'un programme

Exercice 1 Traitement des variables

1. Donner, a priori, l'emplacement des variables du programme suivant (pile, tas, données constantes, ...).
2. La commande `size` permet de déterminer la taille des différents segments d'un exécutable. Compiler le programme en commentant certaines variables et lancer la commande `size` pour vérifier la taille des segments et déterminer où sont placées les variables.
3. Décrire l'évolution de la pile au cours de l'exécution du programme (schéma) à partir d'une adresse de base.

```c
#include <stdio.h>

const int ci1 = 5;

const char* cc1 = "Affichage1\n";
const char* cc2 = "Affichage2\n";

int gi1 = 3;
int gi2;

static int sgi1;

int fct1() {

  static int si1 = 5;
  int i, j;

  i = si1 * 2;
  j = si1 - 1;

  si1 += 1;

  return i + j;

}
```

```c
int fct2(int arg) {

  int i = 5;
  int j = i + fct1();

  return i + j;// + ci1;

}

int main() {

  int a, b;

  printf( "%s\n", cc1 );

  a = fct1();
  b = fct2(a);

  printf("a=%d\tb=%d\n", a, b);
  printf( "%s\n", cc2);

  return 0;

}
```

En C, le mot-clé `static` ajouté devant une variable globale change la portée de cette variable qui devient locale au fichier et ne peut donc plus être accédée depuis un autre fichier. Par contre cette variable est toujours stockée dans le même segment. Par défaut les variables globales sans ce préfixe sont considérées comme accessible globalement.

On peut également ajouter le mot-clé `static` devant une fonction pour limiter également sa portée au fichier. Par défaut les fonctions sans ce préfixe sont considérées comme accessible globalement.

Le mot-clé `static` a par contre un effet différent lorsqu'il est placé devant une variable locale.

Exercice 2 Variables locales `static`

1. La compilation du code suivant produit une erreur, expliquer pourquoi ?
2. Décommenter le mot clé `static` devant la déclaration de la variable `i` et expliquer pourquoi la compilation s'effectue correctement. Observer les symboles définis par le programme à l'aide de la commande `nm` avec et sans `static`.
3. Expliquer les résultats affichés à l'exécution du programme. Où est stockée la variable i ?
4. Où sont stockées les autres variables dans le programme ?
5. Générer le code assembleur de votre programme (`gcc -S exo2.c -m32 -masm=intel`). Observer la différence entre la variable `i` et la variable `value`.
6. Renouveler l'opération en enlevant l'option `-m32`. Le code assembleur est alors généré pour l'architecture x86_64 (Celle des processeurs Intel/AMD 64 bits). Quelles sont les différences avec le code assembleur précédent ?

```c
#include <stdio.h>

int* fct(int a) {

  /* static */ int i = 11;

  i += a;

  return &i;

}

int main() {

  int* result;
  int value = 5;

  result = fct(value);
  result = fct(value);

  printf("result=%i\n", *result);

  return 0;

}
```

Organisation mémoire d'un processus

Exercice 3 Observation des segments en mémoire

1. Compiler le programme suivant (toujours avec l'option `-m32`) puis exécuter le. Ne pas appuyer sur `Entrée` pour en sortir. Le programme est alors en attente. Comment récupérer le `pid` du programme ?
2. Qu'affiche le programme ?
3. À partir du `pid` obtenu, afficher l'emplacement des différents segments en mémoire du processus à l'aide de la commande `cat /proc/`*pid*`/maps`. Commenter les différentes informations affichées.
4. À partir de la sortie du programme, organiser les variables dans les différents segments.
5. Indiquer les variables stockées de manière contigüe en mémoire.
6. Comment les variables sont-elles placées dans la pile ?
7. Décommenter la ligne `mes[0] = '*'`. Que se passe-t-il ? Expliquer pourquoi.

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

const int nbMois = 12;
int gi = 5;
int Tab [] = { 1, 2, 3};
char Titre [50];

int main () {

  int   i    = 2;
  int* pti = &i;       // pointeur sur i
  char* mes = "il fait beau";
  char* ad;
  int   T [2];         // tableau de 2 entiers
  char c;
  static int s = 10;
  ad = (char*) malloc (512); // allocation de 512 octets
  strcpy (Titre, "******************************");

  printf("&main=%p\n", &main);
  printf("&nbMois=%p\tnbMois=%d\n", &nbMois, nbMois);
  printf("&gi=%p\tgi=%d\n", &gi, gi);
  printf("Tab=%p\t&Tab[0]=%p\tTab[0]=%d\n", Tab, &Tab[0], Tab[0]);
  printf("Titre=%p\t&Titre[0]=%p\tTitre[0]=%c\n", Titre, &Titre[0], Titre[0]);
  printf("\n&i=%p\ti=%d\t&pti=%p\tpti=%p\t*pti=%d\n", &i, i, &pti, pti, *pti);
  printf("&mes=%p\tmes=%p\tmes[0]=%c\t*mes=%c\n", &mes, mes, mes[0], *mes);
  printf("&ad=%p\tad=%p\tad[0]=%c\t*ad=%c\n", &ad, ad, ad[0], *ad);
  printf("T=%p\t&T[0]=%p\tT[0]=%d\n", T, &T[0], T[0]);
  printf("&c=%p\tc=%c\n", &c, c);
  printf("&s=%p\ts=%d\n", &s, s);

  //mes[0] = '*';

  printf("En attente ... Observer l'organisation memoire du processus cf. feuille de TD.");
  scanf ("%c", &c); // bloque le processus pour examen
  return(0);

}
```