TP 3 : quelques difficultés ou dangers du langage C - l'assembleur inline

---

Les objectifs de ce TP sont 1) de vous familiariser avec les problèmes liés à l'utilisation du langage C et 2) d'exécuter quelques programmes en langage C incluant de l'assembleur.

---

1 Les chaînes de caractères

La gestion des chaînes de catactères en C est assez délicate, notamment en ce qui concerne l'utilisation des types `char *` et `char []`. Nous allons réaliser un exercice pour mettre en évidence ces problèmes.

Soit le code suivant :

```
int main()
{
  char * ch1="une chaine";
  char ch2[]="une chaine";


  strcpy(ch1,"ecrase");
  strcpy(ch2,"ecrase");
  ch1[2]='l';
  ch2[2]='l';
  ch1="autre chaine";
  ch2="autre chaine";

  return 0;
```

```
}
```

Expliquez précisément les différents problèmes de ce code. Vous aurez éventuellement besoin de la commande `objdump -t`.

2 Les zones de mémoire

Il est important d'avoir en tête les différentes zones de mémoire dans lesquelles peuvent se trouver les différentes variables que l'on peut utiliser dans un programme C. L'exercice suivant est destiné à vérifier que vous avez bien compris ces zones et les variables qui peuvent s'y trouver en fonction de leur déclararation en langage C.

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

int i1;
static int i2;

int main()
{
  int j;
  int * p;
  static int k;

  p=malloc(10);

  // completer

  return(0);

}
```

Sans exécuter le programme, quelles sont selon vous les variables qui vont se situer dans les mêmes zones mémoires ? Complétez ensuite le code pour le vérifier.

Exécutez le programme pour vérifier ensuite.

3 Undefined behavior

Ces undefined behavior, étant par définition indéfinis dans le langage C, sont donc soumis au bon vouloir des différents compilateurs, au moment de la génération du fichier binaire. Il peut ainsi en découler des comportements très différents du même programme, en fonction du compilateur qui l'a compilé et des options de compilation choisie. Dans ce TP, nous envisageons quelques exemples de ces undefined behaviors.

3.1 Division par 0

Soit le code C suivant :

```
#include <stdio.h>

int main()
{
  int msize;

  scanf("%d",&msize);
  if (!msize) {
    printf("oulala\n");
    msize=1/msize; // provoke a signal
  }
  return 0;
}
```

Ce code est conçu volontairement pour lever une exception (sur un processeur Intel) en cas où `msize` est null. La division par 0 étant un undefined behavior en C, les compilateurs sont libres de gérer ce code comme ils le souhaitent. Dans le cas d'optimisation de `gcc`, grâce à l'option `-O2 ou -O3`, on constate que la division par 0 est purement supprimée. Faites le test.

3.2 Décalages

Un décalage bit à bit est un undefined behavior si le décalage est égal ou supérieur à la taille en mémoire de la variable décaléee. Ainsi un décalage de 32 bits d'une variable de type `int` est un undefined behavior.

Soit le code C suivant :

```
#include <stdio.h>

int main()
{
  int j;
  scanf("%d",&j);
  unsigned int i=1 << j;
  printf("%x\n",i);
  i=1 << 32;
  printf("%x\n",i);
}
```

En fonction du compilateur utilisé, le résultat de ce programme peut être complètement différent (entre compilateurs et entre options d'optimisation sur le même compilateur). Faites le test et analysez l'assembleur pour comprendre.

3.3 Les entiers signés

Le débordement d'entier signé étant un undefined behavior en langage C. Cela peut amener certains compilateurs à ne pas considérer certains tests et à les supprimer. Prenons l'exemple du code C suivant :

```
#include <stdio.h>

int main()
{
  int x; // saisir 2147483647

  scanf("%d",&x);
  if (x+100 > x)
    printf("oula\n");
  else
    printf("pas oula\n");
  printf("%d\n",x+100);
}
```

En fonction de l'optimisation utilisée par le compilateur, le test peut tout simplement disparaître du code, alors qu'il peut avoir été prévu par le développeur qui pense ainsi éviter tout débordement d'entier. Faites le test.

4 Dépister les dangers

4.1 Exercice 1

Soit le code C suivant, qui cherche à vérifier qu'un numéro de carte bleue saisie comprend bien 8 chiffres et rien d'autre.

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

int main(int argc, char * argv[])
{
  char CB[9];
  char ISBN[11];
  unsigned int nb;
  unsigned int prix;
  int i;
  int sum=0;
  int OK=0;
  unsigned short tab[10]={0,0,0,0,0,0,0,0,0,0};
  FILE * fd;

  strlcpy(ISBN,argv[1],11);
  sscanf(argv[2],"%d",&nb);
  sscanf(argv[3],"%d",&prix);
  strlcpy(CB,argv[4],9);

  // Check CB : 8 numbers exactly and nothing else

  if (strlen(argv[4])==8) {
    for (i=0;i<8;i++) {
      tab[CB[i]-'0']+=1;
    }
    for (i=0;i<10;i++) {
```

```
      sum+=tab[i];
    }
    if (sum==8) OK=1;
  }

  if (OK==1) printf("good CB number !\n");
  else printf("bad CB number !\n");
}
```

Montrez qu'il est possible de fournir un numéro non valide qui sera néannmoins accepté par le programme.

4.2 Exercice 2

Soit le programme C suivant :

```
#include <stdio.h>

#define MAX 20

int * create_tab(int indice, int val)
{
  int tab[MAX];
  int i;

  for (i=0;i<indice;i++)
    tab[i]=val;
  for (i=indice;i<MAX;i++)
    tab[i]=0;
  for (i=0;i<MAX;i++)
    printf("%d ",tab[i]);

  printf("\n");
  return tab;
}

void afficher(int tab[], int taille)
{
```

```
  int i;

  for (i=0;i<taille;i++)
    printf("%d ",tab[i]);
  printf("\n");
}

int main()
{
  int * tab;

  tab=create_tab(10,5);
  afficher(tab,MAX);

  return (0);
}
```

1. Qu'affiche-t-il selon vous ? Justifiez votre réponse.
2. Proposez une correction du code.

4.3 Exercice 3

Soit le programme C suivant :

```
#include <stdio.h>

#define MAX 5

int main()
{
  int tab[MAX];
  int i;

  for (i=0;i<=MAX;i++)
  {
    printf("%d\n",i);
    tab[i]=2;
  }
```

```
  return 0;
}
```

Ce programme, lorsqu'on l'exécute, pourrait entrer en boucle infinie. Expliquez précisément pourquoi cela peut-il arriver. On supposera que le programme s'exécute sur une architecture 32 bits pour simplifier.

4.4 Exercice 4

Pour chacun des trois codes suivants, identifiez comment un débordement de mémoire est possible.

Code 1 :

```
u_char *make_table(unsigned int width, unsigned int height,
                   u_char *init_row)
{
  unsigned int n;
  int i;
  u_char *buf;

  n = width * height;
  buf = (char *)malloc(n);
  if (!buf)
    return (NULL);
  for (i=0; i< height; i++)
    memcpy(&buf[i*width], init_row, width);
  return buf;
}
```

Code 2 :

```
struct header {
  unsigned int length;
  unsigned int message_type;
};

char *read_packet(int sockfd) {
  int n;
```

```
  unsigned int length;
  struct header hdr;
  static char buffer[1024];

  if(full_read(sockfd, (void *)&hdr, sizeof(hdr))<=0){
    error("full_read: %m");
    return NULL;
  }
  length = ntohl(hdr.length);
  if(length > (1024 + sizeof (struct header) - 1)){
    error("not enough room in buffer\n");
    return NULL;
  }
  if(full_read(sockfd, buffer, length - sizeof(struct header))<=0)
  {
    error("read: %m");
    return NULL;
  }
  buffer[sizeof(buffer)-1] = '\0';
  return strdup(buffer);
}
```

Code 3 :

```
char *read_data(int sockfd)
{
  char *buf;
  int length = network_get_int(sockfd);

  if(!(buf = (char *)malloc(MAXCHARS)))
    die("malloc: %m");
  if(length < 0 || length + 1 >= MAXCHARS){
    free(buf);
    die("bad length: %d", length);
  }
  if(read(sockfd, buf, length) <= 0){
    free(buf);
```

```
    die("read: %m");
  }
  buf[length] = '\0';
  return buf;
}
```

5 Utilisation de l'assembleur dans du langage C

Il est possible d'inclure des instructions assembleur dans un programme C. Cela peut être particulièrement utile lorsque l'on désire exécuter du code qui est "très proche" du processeur, par exemple pour récupérer le contenu d'un registre. La syntaxe pour réaliser ce type d'opérations est la suivante.

```
__asm__("instructions assembleur"
        : operandes de sortie des instructions
        : operandes d'entree des instructions
        : modifications sur registres ou memoire
       );
```

Prenons un exemple :

```
#include <stdio.h>
int main(void)
{
 int a = 10;
 int b = 5;

 __asm__("movl %1, %0"

         : "=&r" (a) : "r" (b)
         : /* liste des modifications */
        );

 printf("a = b = %d \n", a);
 return 0;
```

Cet exemple est bien évidemment un exemple d'école puisqu'il n'y a pas besoin d'utiliser de l'assembleur dans ce cas précis mais il permet d'utiliser un exemple assez simple. Dans l'instruction assembleur `movl %1 %0`, les notations `%1 et %0` font respectivement références au second et au premier opérande qui sont précisés dans la suite. L'opérande `%1` est `"r" (b)` et il signifie que l'opérande (d'entrée) sera un registre (`r`) dans lequel sera copié `b`. L'opérande `%0` fait référence à `"=&r" (a)`, opérande de sortie et il désigne un registre également `r` pour stocker la valeur de `a`. La notation `=&r` signifie que les registres pour stocker les valeurs de `a` et `b` doivent être différents.

Les instructions assembleurs (en syntaxe AT&T) peuvent utiliser des registres et des opérandes. Les registres sont préfixés par `%%` et les opérandes sont représentés par le caractère `%` suivi de l'indice de l'opérande. Plusieurs instructions sont séparées par des `;` sauf la dernière.

Les opérandes sont écrits comme une expression C entre parenthèses précédée d'une chaîne de caractères indiquant une contrainte. La chaîne est constituée d'un ou plusieurs modificateurs et lettres. Le modificateur `=` indique que l'opérande est en écriture, `+` indique qu'il est lecture/écriture.

La lettre `r` indique que l'opérande est stocké dans un registre, `m` spécifie un opérande mémoire et `i` spécifie un opérande immédiat.

5.1 Exercice 1

Ecrire un petit programme en langage C qui déclare une variable locale et qui calcule l'offset entre la position en mémoire de cette variable et le pointeur de pile.

5.2 Exercice 2

Soit le code suivant :

```
asm
  ("mov %1, %%eax\n\t"
   "mov %2, %%ecx\n\t"
   "add %%ecx, %%eax\n\t"
   "mov %%eax, %0"
    :"=r"(c)
    :"r"(a), "r"(b)
    :"%eax", "%ecx"
```

```
);
```

— Insérez ce code dans un programme C afin de vérifier ce que ce code fait.
— Mettez en évidence l'utilité de la ligne `clobbered register` en générant et analysant l'assembleur correspondant.

5.3 Exercice 3

Utiliser l'instruction `cpuid` pour :
— Obtenir le type de processeur utilisé sur votre machine
— Obtenir la liste des features de votre processeur (ainsi que vous pouvez le trouver dans le fichier `/proc/cpuinfo` par exemple)
— Afficher la *Processor Brand String*.

La page `https://en.wikipedia.org/wiki/CPUID` vous donnera toutes les informations nécesaires. En particulier, vous vous intéresserez aux cas où `EAX` vaut 0 pour obtenir le type de processeur et le cas où `EAX` vaut 1 pour récupérer les features. Ces features seront stockés dans plusieurs registres dont `EAX` et `EDX`. Pour stocker et afficher ces features, on vous propose d'utiliser un champ de bits pour `EAX` et un simple entier et des décalages pour `EDX`.