TD 2 : Mémoire

Algorithmes et programmation.

2023

Dans tout le TD, on supposera, pour simplifier que l'espace requis en mémoire pour encoder les différents types est le suivant :

Type	Taille (nombre d'octets)
`int`	1
`long`	2
`float`	1
`double`	2
`char`	1
`void *`	1

Exercice 1 — *Quelques exemples*

Représentez ce qu'il se passe en mémoire si on exécute les codes suivants.

1. Manipulations classiques

```
1   void test (int* p, int* r){
2     int a;
3     double b;
4     int *q;
5     a = 42;
6     b = 73.5;
7     p = &a;
8     q = (int*) malloc(sizeof(int));
9     *p = b + a ;
10    *q = *p + 5;
11    *r = *p + *q;
12    q = &a;
13    p = (int*) malloc(sizeof(int));
14    *p = *q + a;
15    *r = *r + *p;
16  }
17  void main(void){
18    int x;
19    test(&x, &x);
20    printf("%p %d", &x, x);
21  }
22
```

2. Chaînes, pointeurs et tableaux.

```c
#define N 3

void test(char** t){
  for(int i = 0; i < N; i++){
    char c[4];
    if(i % 3 == 0){
      c[0] = 'a';
      c[1] = 'b';
      c[2] = 'c';
      c[3] = '\0';
      t[i] = c;
    }
    else if(i % 3 == 1){
      t[i] = "def";
    }
    else{
      c[0] = 'g';
      c[1] = 'h';
      c[2] = 'i';
      c[3] = '\0';
      t[i] = c;
    }
  }
}

int main(void){
  char ** t = malloc(N * sizeof(char*));
  test(t);
  for(int i = 0; i < N; i++){
    printf("t[%d] = %s\n", i, t[i]);
  }
  free(t);
  return 0;
}
```

Comment corriger ce programme ?

3. Structures, pointeurs et realloc.

```c
#define N1 2
#define N2 4

struct s_rec{
  long value;
  struct s_rec * first;
};
typedef struct s_rec rec;

int main(void){

  rec* t = (rec *) malloc(N1 * sizeof(rec));
  for(int i = 0; i < N1; i++){
    t[i].value = 0;
    t[i].first = t;
  }
  malloc(sizeof(int));
  t = realloc(t, N2 * sizeof(rec));
  for(int i = N1; i < N2; i++){
    t[i].value = i;
    t[i].first = t;
  }

  for(int i = 0; i < N2; i++)
    t[i].first->value = 165449;
    *t.value = 134789;

  for(int i = 0; i < N2; i++)
    printf("%d : %d\n", i, t[i].first->value);

  return 0;
}
```

Comment corriger ce programme ?

Exercice 2 — *Défragmentation*

1. On suppose qu'on dispose d'une mémoire de `2N` octets sur le tas (la zone où sont alloués les espaces réservés par `malloc`).
   Que se passe-t-il avec le code suivant ? Représentez le tas en mémoire.

```
int* p[2*N];
for(int i = 0; i < 2*N ; i++){
  p[i] = (int*) malloc(sizeof(int));
}
for(int i = 0; i < 2*N ; i = i + 2){
  free(p[i]);
}
double * q = (double*) malloc(sizeof(double));

```

2. Supposons que le système soit capable de défragmenter : si un `malloc` devait renvoyer `NULL` par manque de place, les pointeurs précédemment alloués sont repositionnés sur le tas de sorte à mettre tout l'espace disponible à la fin.
   Représentez le tas en mémoire à l'issue du code précédent.
3. En supposant la défragmentation effectuée, sur quoi pointe `p[1]`. En déduire que ce modèle de défragmentation n'est pas utilisable dans des langages qui manipulent sans contrainte des variables de type pointeur.
4. On veut implémenter une fonction `malloc2` et une fonction `free2` qui utiliseront `malloc` et `free` et qui défragmentent quand il y a besoin. Pour simplifier on suppose qu'on ne travaille qu'avec des entiers. On fait les hypothèses suivantes :
   - En haut du fichier sont définis, avec un `define`, deux tailles `S` et `P`.
   - On dispose d'un tableau global `t` de `P` pointeurs sur entiers.
   - On dispose d'un tableau global `size` de `P` entiers ; où `size[i]` contient le nombre d'entiers sur lesquels pointe `t[i]`.
   - On dispose d'un entier global `s` qui donne la taille effective de `t` et `size`.
   - Un pointeur de `t` ne peut pointer sur plus de `S` entiers.

   (a) Implantez un algorithme `int* malloc2(int size)` qui alloue la mémoire pour un pointeur sur `size` entiers, si `size <= S`, et ajoute ce pointeur à `t`. S'il n'y a pas la place nécessaire, on appelle une fonction qui défragmente la mémoire avant de réessayer.
   (b) Implantez un algorithme `void free2(int* p)` qui libère la mémoire d'un pointeur allouée par la fonction `malloc2`.

   On fait maintenant les hypothèses suivantes :

   - On suppose qu'aucune autre fonction du programme ne fait appel à `malloc` ou à `free`.
   - On suppose que `malloc` attribue toujours la première place disponible en mémoire : l'adresse la plus petite permettant d'allouer un bloc mémoire de la taille demandée.
   - On dispose d'une fonction `sort` qui trie le pointeurs de `t` de la plus petite à la plus grande adresse.

   (c) Implantez un algorithme `void defragmente()` qui défragmente la mémoire.