TDS: MPC

Exercice 27. [Comparaison générique]L'objet de l'exercice est de proposer une fonction `mmemcmp()` qui compare deux zones mémoire octet par octet.

1. Donnez un prototype possible pour cette fonction.
2. Donnez une définition de cette fonction. La fonction renverra une valeur nulle si et seulement si les deux zones sont égales.

Pour information. La bibliothèque `string.h` propose une fonction `memcmp()` qui réalise une telle comparaison.

```
int mmemcmp(void *z1, void *z2, unsigned int size){
for(int i=0; i<size; i++){
if((char*)z1[i] != (char*)z2[i]){
return 0
}
return 1;
}
```

Exercice 28. [Appliquer une fonction à un tableau d'entiers]
Soit `func_t` le type des fonctions à un paramètre entier qui renvoient une valeur entière :

```
typedef int (func_t)(int);
```

L'objet de la fonction `map_int()` que nous allons écrire est d'appliquer une fonction de type : chacun des éléments d'un tableau d'entiers. Le résultat sera produit dans autre tableau d'entier

1. Étant données cette fonction `map_int()` qui produit dans le tableau `res` l'application de la `f` à chacun des `size` éléments du tableau `arg`:

   ```
   void map_int(const int arg[], int res[], unsigned int size, func_t *f);
   ```

   et les définitions suivantes:

   ```
   #define MAX 128
   int t[MAX], t2[MAX];
   int carre(int n){
      return n*n;
   }
   ```

   En supposant le tableau `t` initialisé, donnez le code d'un appel à `map_int()` pour produire dans `t2` les carrés des valeurs de `t`.

2. Proposez une définition de `map_int()`.

```
int t[50] = {...};
int Res[50];
map_int(t, Res, MAX, CARRE);

int map_int(...){
int i;
for(i=0; i<size; i++){
Res[i] = f(t[i]);
}
Return 1;
}
```

Exercice 29. [Map générique] La fonction `map_gen()` est une version générique de la fonction `map_int()`, capable de traiter des tableaux de type quelconque.
Cette fonction `map_gen()` utilise une fonction de type `funcgen_t` pour appliquer une opération sur chaque élément d'un tableau :

```
typedef void (funcgen_t)(const void*, void*);
```

Le principe d'une telle fonction est de considérer son premier paramètre comme une référence vers un argument, et renvoyer son résultat via le second paramètre.
Le prototype de `map_gen()` est le suivant :

```
void map_gen(const void *arg, void *res, unsigned int nbelem,
             unsigned int elemsize, funcgen_t *f);
```

Les paramètres `arg` et `res` sont des tableaux de `nbelem` élements, chaque éléments occupe `elemsize` octets. La fonction produit la valeur `f(arg[i])` dans l'élément `i` du tableau `res`.

1. Soient les définitions suivantes :

   ```
   #define MAX 128
   int t[MAX], t2[MAX];
   void carre_int_v1(const void *src, void *dst){
     int* src_int = (int*) src;
     int* dst_int = (int*) dst;
     *dst_int = *src_int * *src_int;
   }
   ```

   En supposant le tableau `t` initialisé, donnez le code d'un appel à `map_gen()` pour produire dans `t2` les carrés des valeurs de `t`.

2. Faites de même en utilisant maintenant la fonction suivante :

   ```
   void carre_int_v2(const int *src, int *dst){
     *dst = *src * *src;
   }
   ```

3. Donnez la définition de fonctions et le code permettant de produire un tableaux des carrés de valeur de type `float`.

4. Proposez une définition de la fonction `map_gen()`.

```
void map_gen(...){
char *larg, lres; int i;
larg = (char*)arg; lres = (char*) Res;
for(i=0; i<nbElem; i++){
    funcgen((void*)larg, (void*)lres);
    larg += i * elemsize;
    lRes += i * elemsize}
}
Return;
}
```

8 Implémentation (basique) de notre propre `malloc`

Dans ce TD, nous allons implémenter de deux manières différentes un mécanisme d'affectation dynamique de mémoire. Nous supposerons disposer d'une zone mémoire:

```
#define NULL ((void *) 0)
#define BUFFERSIZE 65536
static char buffer[BUFFERSIZE];
```

Dans les deux exercices, nous souhaiterons coder les deux fonctions suivantes:

```
/* retourne un pointeur generique sur un espace memoire de taille n
   Retourne NULL si cete quantite n'est pas disponible.
   Les zones memoires ne doivent a aucune moment se recouvrir */
void* monmalloc(unsigned int n);

/* libere l'espace memoire associe au pointeur p */
void monfree(void *p);
```

Aucune autre fonction ou partie du code n'a à accéder au tableau `buffer`: ce dernier n'est manipulé que par l'intermédiaire des fonctions `monmalloc` et `monfree`.

Exercice 30. [Implantation rudimentaire du type pile (LIFO)]
Pour commencer, on propose une implantation rudimentaire du type pile: les espaces alloués s'empilent dans le tableau et la désallocation consiste en un dépilement. Ainsi, l'utilisateur doit veiller à ce que les appels de la fonction `monfree` se fassent dans l'ordre inverse exacte de ceux de la fonction `monmalloc`.
Ce type d'implantation simple se base sur les définitions :

```
static char *nextfreebyte = buffer;
```

Le pointeur `nextfreebyte` pointe sur le premier octet non alloué du tableau `buffer`.

1. Pourquoi le pointeur `nextfreebyte` est-il de classe d'allocation `static` ?
2. Implantez les fonctions `monmalloc` et `monfree` basées sur le principe ci-dessus.

```
void *monmalloc(unsigned int size) {
    void* res;                    -sizeof(int)
    if ((TOTSIZE - (buffer - next)) <size) {
        return NULL;
    }
    Res = (void *) nextFree;
    nextfree += size;
    (int *)nextfree = size; nextfree += sizeof(int);
    Return Res;
}
void free (void) {
    unsigned int size;
    size = *((int*) nextfree - 1);
    nextfree -= size + sizeof(int);
    Return;
}.
```