TP Systèmes 1 : Allocateur mémoire

Ensimag 2A, édition 2014-2015

1 Allocation Mémoire

Le système que nous nous proposons d'étudier possède un espace mémoire de taille maximale fixée (constante définie par 2^{20} octets). Cet espace correspond à la mémoire physique utilisable par le système ou par l'utilisateur. Le problème qui se pose est de fournir un mécanisme de gestion de la mémoire.

La gestion de la mémoire signifie :

— connaître les zones mémoires utilisées ainsi que celles libres, i.e. disponibles pour le système ou l'utilisateur lorsqu'ils en demandent l'allocation,

— allouer des zones mémoires libres lorsque le système, ou l'utilisateur, les demande,

— libérer des zones mémoires utilisées lorsque le système, ou l'utilisateur, les rend.

2 Les variantes

Vous devez choisir le sujet précis. Il faut indiquer son numéro dans le script cmake (cf. sec 4.3).

Pour cela, il faut modifier dans `CMakeLists.txt` la ligne `set(VARIANTE_LOGINS login1 login2 login3)` pour y mettre les logins de l'équipe à la place de `login1 login2 login3`. et `set(VARIANTE_SUJET -1)` pour y mettre votre numéro de variante du sujet.

ID	Variantes
0	Premier d'abord, chaînage circulaire (cff) (sec. 3.2)
1	Algorithme du compagnon (buddy) (sec. 3.1)
2	Algorithme du compagnon pondéré (buddy) (sec. 3.3)

Table 1 – Les différentes variantes

3 Mise en œuvre

3.1 Algorithme du compagnon (buddy)

Dans les méthodes d'allocation par subdivision, les tailles de mémoires allouées sont quantifiés : elles sont exprimées en multiples d'une certaine unité d'allocation et les tailles permises sont définies par une relation de récurrence. Deux systèmes usuels sont :

— le système binaire (1,2,4,8...)

— le système Fibonacci (1,2,3,5,8,13...)

Nous allons utiliser le système binaire.

Pour chacune des tailles, une liste des blocs libres est conservée dans une des cases d'une table appelée "Table des Zones Libres" (TZL). Dans le système binaire, il y a donc une liste des blocs de taille 2^0 dans la case 0, 2^1 dans la case 1, 2^2 dans la case 2, ...2^i dans la case i.

Les informations sur les zones libres sont stockées dans les zones libres elle-même, en particulier le pointeur vers le suivant dans la liste.

Lorsque l'on manque de blocs d'une certaine taille S_i, on subdivise un bloc de taille S_{i+1}. Dans le système binaire, un bloc de 8 est donc divisé en 2 blocs de 4. ces deux blocs issus d'un même bloc original sont dit "compagnons" ("buddy" en anglais). En pratique, pour chaque bloc, l'adresse de son compagnon est fixe.

Ce mode d'allocation a 2 avantages :

— la recherche d'un bloc libre de la bonne taille est rapide

— trouver le compagnon d'un bloc est facile puisque l'on peut calculer son adresse et donc le chercher dans la liste des zones libres de la même taille que le bloc libéré.

3.2 Algorithme premier d'abord, chaînage circulaire (circular first fit)

Dans les méthodes d'allocation par *first fit*, les tailles de mémoires allouées sont quelconques. Une liste chaînée des blocs libres est conservée. Chaque bloc libre contient sa propre taille ainsi qu'un pointeur vers le bloc libre suivant. La liste est circulaire, le dernier bloc pointant sur le premier bloc de la liste.

Il faut noter que les blocs libres doivent pouvoir contenir les informations et donc qu'ils ont une taille minimale. Le bloc choisi pour une allocation est le premier bloc libre dont la taille est plus grande que la taille demandée. À sa libération, un bloc devra être fusionné immédiatement avec son ou ses voisins si ils sont libres.

L'avantage de cette implantation est la simplicité. Les inconvénients sont le fractionnement de la mémoire et les piètres performances puisque l'allocation et la libération d'un bloc demandent le parcours de la liste.

Quelques détails :

— La taille d'une zone sera stockée dans un type entier `unsigned long` (de la même taille qu'un pointeur),

— Comme il n'est pas possible de stocker des blocs libres de tailles inférieures aux informations à stocker, il faut faire attention aux arrondis dans la découpe. Le plus simple est de ne travailler que sur des blocs multiples de cette taille minimale.

3.3 Algorithme du compagnon pondéré (weighted buddy)

L'algorithme est semblable à l'algorithme du buddy (cf section 3.1), mais les blocs sont des tailles 2^k et 3×2^k. Cela double le nombre de tailles manipulées. En effet entre les tailles 2^k et 2^{k+1}, on aura la taille $3 \times 2^{k-1}$.

Lorsque l'on coupe un bloc de taille 2^k, il est coupé en blocs de taille 2^{k-2} et $3 \times 2^{k-2}$. Lorsque l'on coupe un bloc de taille 3×2^k, il est coupé en blocs de taille 2^k et 2^{k+1}.

La TZL aura donc alternativement les deux types de tailles.

Nous vous conseillons de découper les blocs en utilisant les adresses basses pour le petit compagnon (le compagnon gauche) et les adresses hautes pour le grand (le compagnon droit).

Pour l'implantation du découpage et de la fusion, nous vous conseillons d'utiliser deux tableaux supplémentaires `SIZE[idx]` et `SUBBUDDY[idx]` qui indique respectivement la taille des blocs de l'indice *idx* de la TZL et l'indice dans la TZL du plus petit des deux compagnons de découpe (le gauche). La taille du petit compagnon est `SIZE[SUBBUDDY[idx]]`. Celle du grand est `SIZE[idx-1]`.

La relation lors de la découpe d'un bloc correspondant à la ième entrée de la TZL est donc :

```
SIZE[idx] = SIZE[SUBBUDDY[idx]] + SIZE[idx-1]
```

Pour le calcul de l'adresse du compagnon, nous vous conseillons d'implanter une fonction de recherche dichotomique qui utilise les tableaux `SIZE`, `SUBBUDDY` et l'adresse du bloc libéré `adr`.

Le coeur de la fonction ressemblera à

```
while ( idx_cour > idx_cible - 3) {
    if ( min == adr && idx_cour == idx_cible )
      break;
    if ( ( min + SIZE[SUBBUDDY[idx_cour]] ) <= adr ) {
```

```
        adr_buddy = min;
        buddysize = SIZE[SUBBUDDY[idx_cour]];
        min =  min + SIZE[SUBBUDDY[idx_cour]];
        idx_cour = idx_cour - 1;
      }
      else {
        buddysize = SIZE[idx_cour -1];
        max =  min + SIZE[SUBBUDDY[idx_cour]];
        adr_buddy = max;
        idx_cour = SUBBUDDY[idx_cour];
      }
}
```

Elle recalcule la façon dont le bloc a été découpé. Cette fonction a un coût d'exécution logarithmique en la taille de la mémoire.

4 Travail demandé

Il est demandé de réaliser le gestionnaire d'allocation mémoire avec l'algorithme correspondant. Pour cela, vous fournirez les fonctions suivantes :

- `int mem_init()` : alloue la zone de mémoire (c'est le seul malloc autorisé dans votre code) et réalise l'initialisation des structures de données utilisées par le gestionnaire d'allocation de la mémoire. L'adresse de la zone sera conservée dans la variable globale `zone_memoire`. La fonction retournera un code d'erreur indiquant si l'initialisation s'est bien passée ou non. Elle renvoie 0 si tout s'est bien passé.
- `int mem_destroy()` : libère toutes les structures et la zone de mémoire utilisées.
- `void *mem_alloc(unsigned long tailleZone)` : allocation d'une zone mémoire initialement libre de taille tailleZone. La fonction retournera le pointeur vers cette zone mémoire. Le retour sera (void *)0 en cas d'erreur ou s'il n'existe plus d'emplacement libre de taille tailleZone.
- `int mem_free(void * zone, unsigned long tailleZone)` : libère la zone commençant à l'adresse zone de taille tailleZone (valeur renvoyée par `mem_alloc`). Un retour d'erreur sera retourné en cas de problème, sinon 0 sera retourné si tout s'est bien passé.

`int mem_init()` et `void mem_destroy()` peuvent être appelées de nombreuses fois, l'une après l'autre, dans l'ordre.

Ces fonctions seront réalisées au sein du module mem (défini par le fichier `mem.c` et `mem.h`). Ces fonctions seront celles utilisables par le système ou l'utilisateur. Il vous est ensuite possible de définir des fonctions intermédiaires de travail *non exportées* (i.e. implémentées en `static` et non déclarées dans le `.h`) aux utilisateurs du module mem.

Il vous faudra aussi définir les structures de données si besoin.

Vous n'êtes pas autorisé à utiliser les allocations mémoire de C (malloc et free) sauf pour l'allocation et libération de la zone mémoire gérée.

4.1 Rendu des sources

L'archive des sources que vous devez rendre dans `Teide` est généré par le makefile créé par cmake :

```
cd ensimag-allocateur
cd build
make package_source
```

Il produit dans le répertoire `build`, un fichier ayant pour nom (à vos login près) `Allocphy-1.0.login1;login2;login3-Source.tar.gz`.

C'est ce fichier tar qu'il faut rendre.

4.2 Adressage 32 et 64 bits

Votre code devra compiler et tourner sur des machines 64 bits. Les types de base et les pointeurs n'étant pas toujours de la même taille, il vous faudra donc utiliser `sizeof()` pour obtenir la taille d'un type de manière portable, ou la fixer avec `#include <stdint.h>`.

4.3 Compilation et test unitaires

Le squelette fourni inclut l'utilisation de `cmake` pour construire automatiquement les Makefile utiles.

Vous devez modifier le début du fichier `CMakeLists.txt` pour y insérer vos logins.

La création des Makefile s'effectue en utilisant `cmake` dans un répertoire où seront créés les fichiers générés. Le répertoire "build" du squelette sert à cet usage. Tout ce qui apparaît dans "build" pourra donc être facilement effacé.

La première compilation s'effectue donc en tapant :

```
cd ensimag-allocateur
cd build
cmake ..
make
make test
make check
```

et les suivantes, dans le répertoire `build`, avec

```
make
make test
make check
```

Une batterie de test vous est fournie pour vous aider à construire plus rapidement un code correct. "make test" devrait donc trouver la plupart des bugs de votre programme au fur est à mesure que vous l'écrivez. Vous pouvez lancer directement l'exécutable `alloctest` pour avoir plus de détails avec "make check", ou lancer individuellement les tests qui vous intéressent `./alloctest --help`.

4.4 Shell interactif

Un petit shell interactif `memshell` vous est fourni. Il est utilisable seul, mais il permet aussi de réaliser des tests en utilisant votre débogueur pour inspecter l'état de votre allocateur, par exemple l'état de la TZL.

```
gdb ./memshell
break mem_alloc
layout
run
alloc 64
print TZL
```