Gestion de la mémoire dynamique

Principes

L'allocation dynamique de la mémoire est une fonctionnalité fondamentale qui fait partie des systèmes d'exploitation depuis les années 1960.

La zone mémoire disponible pour faire des allocations dynamiques peut être vue comme un ensemble de blocs mémoire (zone de mémoire contiguë de taille variable). l'allocateur doit répondre à des requêtes de type réservation/libération de certains nombres de ces blocs.

Afin de répondre à ces requêtes, un allocateur doit connaitre l'état de ces blocs: quels sont les blocs libres et ceux occupés (pas encore libérés). Le but d'un allocateur est de répondre à ces requêtes en minimisant d'une part, l'espace mémoire perdu dû à la fragmentation des blocs et d'autre part, le temps de réponse (minimiser l'overhead de gestion).

Il existe plusieurs stratégies pour atteindre ces deux buts mais il n'y a pas de stratégie optimale car cela dépend du comportement du demandeur (les tailles demandées, la durée de vie d'une réservation, l'ordre des libérations, etc...).

La stratégie d'allocation que nous vous proposons d'implémenter est de type First-Fit. Dans cette stratégie, l'allocateur cherche parmi les blocs libres celui qui a une taille supérieure ou égale à la taille demandée. Il arrête sa recherche dès qu'il en trouve un. Si la taille du bloc trouvé est supérieure à la taille demandée (d'une certaine quantité), il soustrait la taille en trop et en fabrique un nouveau bloc libre. Enfin, il marque le bloc trouvé comme occupé et rend son adresse. Vous pouvez dans un second temps implémenter la stratégie Next-Fit. Dans cette stratégie, l'allocateur utilise un pointeur sur le premier bloc libre afin d'accélérer la recherche.

La figure suivante illustrent un exemple d'une allocation de 128 octets.

Lors d'une libération d'un bloc, l'allocateur essaie de fusionner (merger) les blocs libres qui suivent ce bloc (et pas ceux qui précèdent). Les figures 2 et 3 illustrent ce fusionnement de blocs en mettant en évidence la condition d'arrêt : plus de blocs libres contigus après le bloc libéré.

Nous avons vu, lors d'une allocation, que l'allocateur peut-être mené à partitionner un bloc libre si sa taille est strictement supérieur à la taille demandée de certaine quantité. Le choix de cette quantité est très important pour la réutilisation des blocs.

L'allocateur doit définir une taille minimum acceptable au dessous du laquelle le partitionnement ne se fera pas. Autrement dit si la taille restante d'un bloc libre n'est pas significative ce n'est pas la peine de partitionner ce bloc sous peine d'accroitre la fragmentation de la mémoire.

L'allocateur voit les blocs comme multiple de sous blocs de taille fixe, c'est la taille minimum pour répondre à une allocation mémoire. Par exemple : si la taille d'un bloc minimal (un sous bloc) est de 32 octets alors une allocation de 52 octets se traduit par une réservation d'un bloc de 2 * 32 soit de 64 octets. Une allocation de 4 octets se traduit par réservation d'un bloc de 1 * 32 octets. En fait, dans le but de mieux tirer profit du cache de données de niveau 1, on fixe la taille minimale d'un bloc à taille d'une ligne de cache. dans notre cas la taille minimale sera de 64 octets (16 mots de 4 octets).

Lors de la libération, l'allocateur reçoit uniquement l'adresse du bloc à libérer mais pas sa taille. l'allocateur doit mettre en place un mécanisme pour retrouver la taille d'un bloc à partir de son adresse.

L'allocateur associe à chaque bloc un descripteur qui indique l'état du bloc (libre ou occupé) et sa taille. Un bloc est donc composé de deux parties, un descripteur et un champ données. la figure suivante illustre cette association.

Définition des structures de données

Nous allons représenter l'allocateur mémoire et la description de la zone mémoire à gérer par la structure heap_s voici sa définition en C. Le lock permet de séquencialiser les accès concurrents. Pour le test de votre API vous utiliserez un spinlock des PThreads.

```
struct heap_s
{
    spinlock_t lock;
    uint32_t *start;
    uint32_t *limit;
};
```

La structure d'un descripteur de bloc est représentée par la structure block_info_s voici sa définition en C :

```
struct block_info_s
{
     uint32_t busy:1;
     uint32_t size:31;
} __attribute__ ((packed));
```

A l'état initial, la zone mémoire est constituée d'un seul bloc libre.

L'API de notre allocateur est la suivante :

```
error_t heap_init(struct heap_s *heap,
                          uint32_t *start,
                          uint32_t *limit);
void* heap_malloc(struct heap_s *heap,
                          size_t size);
void  heap_free  (struct heap_s *heap,
                          void *ptr);
```

Travail demandé

Le travail à réaliser est d'implémenter cette API et la tester par une application de test.

le contexte de l'allocateur est representé par la structure heap_manger_s, les fonctions de l'allocateur peuvent opérer sur plusieurs contextes. Autrement dit, On peut gérer plusieurs zone mémoire avec les mêmes fonctions de l'API heap_manager à condition de disposer d'autant de descripteur heap_manger_s.

Votre application de teste doit déclarer deux variables globales, une de type heap_s et l'autre est un tableau d'octets représentant la mémoire à gérer par le heap_manager.

```
#define ZONE_SIZE      8*1024

int8_t zone [ZONE_SIZE];
struct heap_s heap;
```

Vous devez donc écrire les trois fonctions de l'API dans le fichier heap_manager.c et leur prototype dans le fichier heap_manager.h. En cas d'échec du malloc, vous devrez faire un parcours de la zone mémoire et fusionner les zones libres contiguës.

Vous écrirez un programme de test dans le fichier main.c mettant en évidence les cas de fonctionnement.

Afin de mettre au point votre allocateur, il est utile d'écrire une fonction qui affiche l'état des blocs de l'allocateur, par exemple

```
void heap_print(struct heap_s *mgr);
```