Riot In Mem Block #9

projet S1 proposé par Georges Eyrolles

georges.eyrolles@enseirb-matmeca.fr, bureau : S216

Le sujet de ce projet vous propose de réaliser la gestion d'une ressource mémoire supplémentaire "#9" disponible à l'exécution d'un programme.

Comme pour toutes ressources, l'objectif d'un programmeur est d'écrire son code en utilisant au mieux cette mémoire (elle est forcément limitée en taille). Pour la mémoire, ce mécanisme est désigné sous le terme d'allocation dynamique.

La mémoire "#9" est accessible au programmeur à travers les fonctions de la bibliothèque "Riot In Mem Block #9"[1]. Elle fournit un service de réservation/libération de cette mémoire (ou d'une partie de cette mémoire) et un service de surveillance (interrogation de l'état courant). La spécification des fonctions de cette bibliothèque vous est fournie (annexe A).

Votre travail consiste à :
- mettre au point les algorithmes de gestion et surveillance ;
- les réaliser en langage C ;
- fournir une série de jeux de tests qui permet de vérifier le comportement des choix effectués (voir un exemple en annexe B).

Pour le délivrable, vous n'avez pas à rendre un seul exécutable comme dans une application standard mais plutôt un ensemble d'exécutables qui correspond à votre série de tests de la bibliothèque.

1 Gérer "#9"

La mémoire disponible à l'exécution pour un programme est assimilable à N cases contiguës (ou séquentielles). En informatique, c'est un tableau de taille N avec une taille de case égale à un octet. La mémoire "#9" est donc représentée par un tableau de type **char** avec une taille fixée à la compilation.

Toutes les données manipulées par un programme ne sont pas forcément stockées dans une seule case mémoire. Cela dépend de la taille en octet des données (en langage C voir la taille d'un **short** et d'un **float** avec l'opérateur **sizeof**).

Il est nécessaire de manipuler, dans ce tableau, des blocs de cases contiguës. Un bloc mémoire "memory block"est alors caractérisé par une adresse de début (adresse du bloc) et une taille en octet.

La mémoire "#9" s'organise en deux ensembles :

"in-use memory block" l'ensemble de blocs de mémoire utilisés par le programme,

"free memory block" l'ensemble de blocs de mémoire libres.

Ces ensembles peuvent être représentés de deux manières :
- par deux tableaux de taille fixée à la compilation,
- par un chaînage des blocs directement dans la mémoire "#9".

1. d'après "riot in cell block #9" chanson écrite par Jerry Leiber et Mike Stoller en 1954 interprétée par The Robins (ou The Coaster). Vous trouverez en vidéo la reprise de cette chanson par le groupe "Dr Feelgood" en 1976.

Après l'organisation de la mémoire, décrivons maintenant le service de gestion.

1.1 Réserver

La demande d'un bloc de mémoire se fait par appel à la fonction :

```
void *rimb9_in_riot(size_t size, char* file, int line);
```

Cette fonction recherche dans les blocs libres un espace suffisant pour répondre à cette demande. Elle retourne l'adresse du début de ce bloc. Ce bloc de mémoire devient un bloc en utilisation. Il est initialisé octet par octet à une valeur spéciale.

Pour les blocs en utilisation, nous conservons l'adresse de début (le pointeur), la taille, des informations sur la demande (nom du fichier et numéro de ligne de l'appel) et un entier indiquant le numéro d'ordre de la demande.

Ce bloc est ajouté à l'ensemble des blocs en utilisation et l'ensemble des blocs libres est lui aussi mis à jour.

problèmatique : Pour répondre à cette demande, un bloc libre peut être divisé en deux blocs. Ce découpage doit éviter d'aboutir à un ensemble de petits blocs. Pour cela, il est parfois raisonnable de fournir un bloc avec une taille réelle supérieure à la taille demandée (mais pas trop quand même !!).

1.2 Libérer

La libération d'un bloc en utilisation se fait par la fonction :

```
int rimb9_out_riot(void* mb);
```

Ce bloc est supprimé de l'ensemble des blocs en utilisation et ajouté à l'ensemble des blocs libres. Le bloc libéré est réinitialisé octet par octet à une valeur spéciale. Les informations conservées pour un bloc libre sont l'adresse de début et la taille. Cette fonction retourne une erreur si l'adresse ne correspond pas à un bloc mémoire en utilisation (adresse de début d'un bloc).

problèmatique : Après plusieurs libérations, la mémoire "#9" peut se retrouver morcelée en plusieurs blocs libres contigus c'est à dire pouvant être rassemblés dans un seul bloc libre. Cette opération de compactage peut s'effectuer au moment de la libération d'un bloc ou bien dans rimb9_in_roit s'il n'exite plus de bloc libre de la taille souhaitée.

2 Surveiller "#9"

2.1 Contrôler les blocs "Blocks Wardens"

Pour surveiller les ensembles de bloc, les fonctions de la bibliothèque fournissent :

- rimb9_nb_free() le nombre de blocs libres, rimb9_nb_inuse() le nombre de blocs en utilisation ;
- rimb9_printinfo_insuse() affiche les informations sur l'ensemble des blocs en utilisation et des blocs libres rimb9_printinfo_free().
- rimb9_getinfo() les informations pour un bloc en utilisation ;

Ces fonctions vous serviront dans l'écriture des tests pour vous assurer de la cohérence de votre réalisation.

2.2 Vérifier la mémoire "Memory Wardens"

En supplément, des fonctions peuvent être ajoutées pour vérifier l'utilisation de la mémoire par le programmeur :

- rimb9_contains_insue() vérifier si une adresse se trouve bien à l'intérieur d'un bloc en utilisation ;
- rimb9_examine_freed() vérifier que les octets des blocs de mémoire libres n'ont pas été modifiés ;

3 Remarques sur le développement

Il ne s'agit pas de prendre en charge tous les problèmes à la fois dès le début. Le sujet offre plusieurs alternatives pour vous permettre d'adapter ce travail à votre niveau en A.S.D. et en programmation C. Au cours du développement, ce niveau va progresser ainsi que votre maîtrise du sujet. Pendant les 8 semaines du projet, votre équipe doit envisager plusieurs versions de cette bibliothèque.

Voici la version la plus directe :
- l'ensemble de blocs est représenté par un tableau ;
- la fonction rimb9_in_riot fournit le premier bloc libre avec une taille exacte par découpage d'un bloc libre si nécessaire ;
- aucun compactage des blocs libres ;
- pas d'initialisation du bloc libéré ou en utilisation ;
- les fonctions "Memory wardens"ne sont pas codées.

Ensuite, vous pouvez prendre en charge les autres problèmes :
- le chaînage des blocs dans la mémoire "#9" ;
- la recherche d'un bloc libre adéquat (division d'un grand bloc ou fournir un bloc d'une taille plus grande) ;
- compactage des blocs libres ;
- réalisation des autres fonctionnalités ...

Ce sujet a été établi en consultant la documentation des fonctions malloc/free de la bibliothèque standard du langage C mais aussi la documentation de la bibliothèque open-source de débogage mémoire `Dmalloc` (http://*dmalloc.com*).

4 Les Annexes

A Le fichier en-tête

Le fichier rimb9.h :

```
#ifndef RIMB9_H
#define RIMB9_H

#include "stddef.h"

//———————— Memory Management ————————
/*
 * request a initialized memory block in number 9 with a least size bytes
 * of storage available.
 * All bytes in this memory block are initialized with a default value.
 *
 * parameters: size memory block size in bytes
 *             file file name of the request
 *             line line number of the request
 *
 * return: start address if block is available otherwise NULL.
 */
void *rimb9_in_riot(size_t size, char* file, int line);

/*
 * release in-use memory block at address mb that was previously
 * request by rimb9_int_riot().
 *
```

```
 * parameter: mb start address of in-use memory block
 *
 * return: 0 if sucessful
 *         1 if mb is not a valide in-use block
 *         2 if not in number 9.
 */
int  rimb9_out_riot(void *mb);

/*
 * set the in-use init value.
 * In rimb9_in_use(), all bytes in the in-use block
 * are initialized with this special value.
 *
 * parameter: v init value.
 */
void rimb9_set_init_inuse(char *iv);

//---------- Blocks Wardens ----------
struct info_inuse_mb {
  size_t size;
  int line;
  char *file;
  long order;
};

/*
 * informations for one in-use memory block
 * parameter: mb start address in-use memory block
 *
 * return: order > 0  if mb is a valid in-use block
 *         otherwise order == 0
 */
struct info_inuse_mb rimb9_getinfo(void *mb);

/*
 * the number of in-use memory block in number 9
 */
int  rimb9_nb_inuse();

/*
 * print all in-use memory block in number 9
 */
void rimb9_printinfo_inuse();

/*
 * the number of free memory block in number 9
 */
int  rimb9_nb_free();

/*
 * print all free memory block in number 9
 */
void rimb9_printinfo_free();


//---------- Memmory Wardens ----------
/*
 * search if a in-use block contains this address.
 * If this address is a start address of a in-use block
 * the search is successful.
 *
```

```
 * parameter: add address to check.
 *
 * return: start address of in-use block if successful
 *         otherwise NULL.
 */
void *rimb9_contains_inuse(void *add);

/*
 * search overwriting in free blocks.
 * The function compare each byte in free block
 * with the freed init value.
 *
 * return: number of overwriting.
 */
int  rimb9_examine_freed_mb();

/*
 * set the freed init value.
 * In rimb9_out_riot(), all bytes in freed block are initialized
 * with this special value.
 *
 * parameter: fv init value.
 */
void rimb9_set_init_freed(char *fv);

#endif /* RIMB9_H */
```

B Un exemple possible de test

Le fichier test-example.c :

```
#include "stdlib.h"

#include "rimb9.h"

int main() {
  double *foo;
  short *doo;

  foo = rimb9_in_riot(5 * sizeof(double), __FILE__, __LINE__);
  foo[2] = 1.89;

  doo = rimb9_in_riot(8 * sizeof(double), __FILE__, __LINE__);
  doo[1] = 3;

  printf("nb bloc en utilisation doit être : 2 -> %d\n", rimb9_nb_inuse());
  printf("nb bloc libre doit être : 1 -> %d\n", rimb9_nb_free());

  rimb9_out_riot(foo);

  printf("nb bloc en utilisation doit être : 1 -> %d\n", rimb9_nb_inuse());
  printf("nb bloc libre doit être : 2 -> %d\n", rimb9_nb_free());

  return EXIT_SUCCESS;
}
```