GUIDE DU DÉVELOPPEUR

Paul Gasquet - Zacharie Baril

UNIVERSITE DE CERGY PARIS L3-I

2020-2021

Table des matières

1. Compilation et documentation

Le dossier où se situe nos programmes contient aussi un « Makefile » qui peut générer les 2 programmes ainsi que la documentation, en tapant dans le terminal :

- « make demo » pour générer l'exécutable du programme de démonstration
- « make test » pour générer l'exécutable du programme de test
- « make doc » pour générer la documentation HTML sur les 2 programmes

2. Structures utilisées

2.1 Liste chainée des blocs de mémoires

```c
/**
 * @struct memoryBlock initializer.h
 * This struct will use for to manage our memory space with memory blocks.
 */
typedef struct memoryBlock{
    /** size of block of memory*/
    size_t size;
    /** if the block memory is free, free = 1, else free = 0 */
    int free;
    /** pointer on the next block */
    struct memoryBlock *next;
    }*BlockListe;
BlockListe bl;
```

La structure ci-dessus permet l'allocation mémoire, elle est implémentée à la manière d'une liste chainée avec des blocs de mémoire **memoryBlock,** ainsi que la taille du bloc de mémoire alloué (ou libérer) **size**, une variable **free** permettant de savoir si un bloc de mémoire est alloué ou non et d'un pointeur sur le bloc de mémoire suivant **next**.

2.2 Structure servant à utiliser notre BlockListe

```c
/**
 * @struct BlockTable initializer.h
 * This struct list memory blocks.
 */
typedef struct{
    /** Table of memory block used during the test programm*/
    struct memoryBlock **block;

    /** Index max of table */
    int indexMax;
} BlockTable;

BlockTable *bt;
```

La structure ci-dessus permet de stocker un tableau de blocs de mémoire avec **block**, qui a un index maximum **indexMax** que l'on incrémente à chaque nouveau bloc crée et que l'on décrémente à chaque fusion de blocs.

3. Allocation mémoire

3.1 Initialisation mémoire

```c
/**
 * Initialize the shared memory zone, in order to create our work zone,
 * and initialize the manager of memory Block.
 * @param nBytes size of zmemory one to allocate
 * @return int the number of bytes effectively allocated on success, 0 on failure
 */
int initMemory(int nBytes);
```

La fonction **initMemory** permet d'allouer la mémoire qui va être utilisée lors de l'exécution du programme. On commence par appeler la fonction **allocTabP** en lui mettant le nombre d'octets avec **nBytes** en paramètre, puis on initialise la structure **BlockListe** avec le tableau de caractère alloué.

```c
/**
 * @brief Allocate the memory for one block given in parameter.
 *
 * @param n size of one block memory
 * @param tab address of block to allocate
 * @return int size in bytes from block effectively allocate
 */
int allocTabP(int n, char **tab);
```

Ici, on alloue toute la mémoire du tableau de caractères **sharingSpaceMemory** avec **n** caractères.

3.2 Allocation dans notre zone mémoire

```c
/**
 * @brief Allocate a memoryblock in the blocklist
 *
 * @param nBytes number of bytes to allocates
 * @return void* address of beginning chain allocate
 */
void* mymalloc(int nBytes);
```

La fonction **mymalloc** gère l'allocation mémoire des blocs de mémoire dans la **blockListe**, la stratégie utilisée est *« best fit »*. Cette stratégie repose sur le fait de trouver dans la blockListe un bloc de mémoire ayant la plus petite taille possible pour faire l'allocation.

Pour ce faire on la parcourt tant que nous ne sommes pas arrivés à la fin de la liste :

- Si lors du parcours on a un bloc de mémoire qui n'est pas alloué et dont la taille en octets est égale à la taille de la donnée que l'on veut allouer alors c'est bon, on allouera ce bloc.
- Sinon si, on trouve un bloc non alloué et qui est assez grand pour l'allocation, on le garde en mémoire jusqu'à ce qu'on en trouve un ayant une taille encore plus proche de la donnée à allouer jusqu'à ce qu'on trouve un bloc parfait ou que la blockListe soit parcourue dans son intégralité.
- Sinon, à trouver aucun bloc libre assez grand alors on précise que la mémoire disponible est insuffisante.

Il faut également préciser que dans le cas d'une allocation d'un bloc n'ayant pas exactement la taille de la donnée à allouer on utilise la fonction **splitAllocate** :

```c
/**
 * @brief Allocate a memory block with the creation of a new memory
 * block who do exactly the requested size.
 *
 * @param currentBlock address of current block
 * @param size of new block
 */
void splitAllocate(struct memoryBlock *currentBlock, size_t size);
```

Cette fonction permet d'allouer un bloc de mémoire faisant la taille exacte de la donnée à alloué, en créant un bloc de mémoire (non alloué) supplémentaire avant le bloc qu'on le souhaite alloué. Ce bloc supplémentaire fait donc exactement la taille superflue par rapport à la donnée à allouer qui était présente dans le bloc (libre) d'origine.

4. Libération de mémoire

4.1 Libération mémoire dans notre espace

```c
/**
 * @brief Free a memoryblock in the blocklist.
 *
 * @param p zone of memory who are going to be deallocated
 * @return int -1 if an error occurs the size of the space deallocated if no problem occurs
 */
int myfree(void *p);
```

La fonction **myfree** permet de libérer un bloc mémoire de la blockListe. Pour ce faire on regarde si le pointeur correspondant à l'objet que l'on souhaite supprimer est bien contenu dans les adresses présentes dans la blockListe. Si oui on vérifie que l'objet à cette adresse n'est pas déjà libérer. S'il ne l'est pas on le libère (par l'intermédiaire d'une variable boolean free) et on appelle la fonction **myrealloc**.

```c
/**
 * @brief Merge memory blocks if we have deallocated 2 memory block odds by odds.
 *
 */
void myrealloc();
```

La fonction myrealloc est utile dans le cas ou 2 blocs de mémoire ont été libérés alors qu'ils étaient situés à deux adresses contiguë dans la blockListe. Si c'est le cas, on prend le bloc de mémoire courant et on place son pointeur suivant vers son prochain voisin qui n'a pas été libérer, ou si ce sont les 2 derniers blocs de la BlockList, alors on fait pointer le nouveau bloc vers NULL.

Cela permet de gérer la **fragmentation** dans la blockListe.

4.2 Libération de toute notre mémoire

```c
/**
 * Free the memory initially allocated with the "initMemory()" function.
 * @return int the number of bytes effectively free on success, -1 on failure
 */
int freeMemory();
```

La fonction **freeMemory** permet de libérer la zone mémoire utilisée, elle est appelée à la fin de l'exécution du programme, en appelant la fonction **libMemTabP**. Puis, on récupère le nombre d'octets totaux libérés renvoyé.

```c
/**
 * @brief Free one block of memory.
 *
 * @param tab address of block to free
 * @return int size of bytes of block effectively free
 */
int libMemTabP(char **tab);
```

Cette dernière prend le tableau ayant servi initialement à l'allocation de la mémoire dans la fonction **initMemory** est utilisé la fonction **free** sur l'adresse de la variable **sharingSpaceMemory**.