Rapport Allocateur Mémoire

fait par : -Kreddia Asma
-Bouymedj Wissam

Table des matières

Allocation mémoire :

Définition :

L'allocation de mémoire vive désigne les techniques et les algorithmes sous-jacents permettant de réserver de la mémoire vive à un programme informatique pour son exécution.
L'opération symétrique de l'allocation est couramment appelée libération de la mémoire (on peut parler également de désallocation ou de restitution).

Support des fonctions d'allocation :

Le but est d'intercepter les fonctions malloc,calloc,realloc et free depuis une bibliothèque tierce, pour cela on a crée les fonctions maMalloc,maCalloc,maRealloc et monFree ,

On a implémenter notre propre Malloc et Free en utilisant respectivement « mmap » et « munmap » qui prennent la longueur comme paramètre, donc on la mis la longueur comme information supplémentaire dans la mémoire mappée et pour ce faire, on a arrondit toutes les tailles souhaitées à la taille de page la plus proche. Ceci est assez inutile, mais économise sur la complexité de l'implémentation.

MMAP pour la fonction Malloc:

Deux fonctions permettent de réserver et de libérer dynamiquement une zone de la mémoire : malloc pour la réservation, dont le prototype est le suivant :

void *malloc(size_t size);

Le seul paramètre à passer à malloc est le nombre d'octets à allouer. La valeur retournée est l'adresse du premier octet de la zone mémoire allouée. Si l'allocation n'a pu se réaliser (par manque de mémoire libre), la valeur de retour est la constante NULL.

Pour implémenter la fonction malloc on utilise mmap ,

mmap() crée une nouvelle projection dans l'espace d'adressage virtuel du processus appelant. L'adresse de démarrage de la nouvelle projection est indiquée dans addr. Le paramètre length indique la longueur de la projection.

Structure :**void *mmap(void ***addr**, size_t** length**, int** prot**, int** flags**,**

int fd**, off_t** offset**);**

sachant que :

PROT_READ :On peut lire le contenu de la zone mémoire.

PROT_WRITE :On peut écrire dans la zone mémoire.

MAP_PRIVATE : Créer une projection privée, utilisant la méthode de copie à l'écriture. Les modifications de la projection ne sont pas visibles depuis les autres processus projetant le même fichier, et ne modifient pas le fichier lui-même. Il n'est pas précisé si les changements effectués dans le fichier après l'appel **mmap**() seront visibles.
MAP_ANONYMOUS

La projection n’est supportée par aucun fichier ; son contenu est initialisé à 0. Les arguments fd et offset sont ignorés ;cependant, certaines implémentations demandent que fd soit -1 si **MAP_ANONYMOUS** (ou **MAP_ANON**) est utilisé, et les applicationsportables doivent donc s’en assurer. Cet attribut, utilisé en conjonction de **MAP_SHARED**, n’est implémenté que depuis Linux 2.4.

et on a definit mmap comme cela :

void * nvregion = mmap(0 ,2000, PROT_READ|PROT_WRITE, MAP_ANON|MAP_PRIVATE, 0, 0);

- 0 est L’adresse de démarrage de la nouvelle projection qui est indiquer dans addr,(donc addr est NULL ici , le noyau choisit l’adresse à laquelle démarrer la projection ; c’est la méthode la plus portable pour créer une nouvelle projection),

- pour la taille on a mis 2000 ,
- la position offset dans le fichier (ou autre objet) correspondant au descripteur de fichier fd est initialisé ici a 0,

MUNMAP pour la fonction Free :

La libération de la mémoire précédemment allouée via malloc est assurée par la fonction free dont la déclaration est la suivante :

void free(void *ptr);

Le seul paramètre à passer est l'adresse du premier octet de la zone allouée et aucune valeur n'est retournée une fois cette opération réalisée.

Pour implémenter la fonction free on utilise mmap ,

munmap() :L’appel système **munmap**() détruit la projection dans la zone de mémoire spécifiée, et s’arrange pour que toute référence ultérieure à cette zone mémoire déclenche une erreur d’adressage. La projection est aussi automatiquement détruite lorsque le processus se termine. À l’inverse, la fermeture du descripteur de fichier ne supprime pas la projection. L’adresse addr doit être un multiple de la taille de page. Toutes les pages contenant une partie de l’intervalle indiqué sont libérées, et tout accès ultérieur déclenchera **SIGSEGV**. Aucune erreur n’est détectée si l’intervalle indiqué ne contient pas de page projetée.
Structure : **int munmap(void *addr, size_t length);**

Pour la fonction maCalloc :

La zone mémoire allouée par `malloc` n'est pas initialisée automatiquement. Cette initialisation peut être réalisée à l'aide de la fonction `memset` ou bien par le parcours de toute la zone mémoire. Avec la fonction `calloc`, cette phase d'initialisation n'est plus nécessaire, car la zone mémoire allouée est initialisée avec des 0.

La déclaration de `calloc` est la suivante :

```
void *calloc(size_t nmemb, size_t size);
```

Le paramètre *nmemb* est le nombre d'éléments que l'on désire réserver et *size* correspond à la taille en octets d'un élément. La valeur retournée est la même que pour `malloc`.

Dans notre programme on l'a générée depuis maMalloc : void* maCalloc(size_t nombre, size_t taille), et en utilisant l'alignas ,Le spécificateur alignas peut être appliqué à la déclaration d'une variable ou d'un membre de données de classe sans champ binaire, ou il peut être appliqué à la déclaration ou à la définition d'une classe / struct / union ou d'une énumération. Il ne peut pas être appliqué à un paramètre de fonction ou au paramètre d'exception d'une clause catch.

L'objet ou le type déclaré par une telle déclaration aura son exigence d'alignement égale à l'expression non nulle la plus stricte (la plus grande) de tous les spécificateurs alignas utilisés dans la déclaration, à moins que cela n'affaiblisse l'alignement naturel du type.

Pour cela on a déclarer en haut #define alignas(x) (((((x)-1)>>2)<<2)+4)

Pour la fonction maRealloc :

La fonction realloc permet de modifier la taille de la mémoire allouée préalablement avec malloc. S'il est nécessaire de déplacer la zone mémoire, car il n'y a pas assez de mémoire contiguë, la libération de l'ancienne zone mémoire est réalisée par realloc via free.

La déclaration de realloc est la suivante :

void *realloc(void *ptr, size_t size);

Le paramètre ptr désigne le début de la zone mémoire dont on désire modifier la taille. Le second paramètre, size, est la nouvelle taille en octet de la zone mémoire.

Si la fonction réussit, la valeur retournée est le début de la zone mémoire allouée. Attention : la valeur du pointeur ptr n'est plus valide car la nouvelle zone peut débuter à un autre endroit de la mémoire si un déplacement a été nécessaire.

Si la fonction échoue, elle retourne la valeur NULL. Des précautions doivent être prises pour éviter une fuite de mémoire. Il convient de veiller à ce que l'adresse de l'ancien bloc ne soit pas écrasée avant de s'assurer que la réallocation a bien réussi.

Dans notre programme on la implémenter comme ce si :void* maRealloc(void * memoire, size_t taille) et on a fait appel a la fonction maMalloc pour pouvoir implementer maRealloc.

Multithreading :

Pour résoudre le problème de concurrence , on a utilisé une mécanisme de synchronisation : Mutex

qui va nous permettre de protéger des données .

on initialise un mutex avec la valeur de la constante PTHREAD_MUTEX_INITIALIZER,

on verrouille le mutex grâce à la fonction : pthread_mutex_lock(&lock),

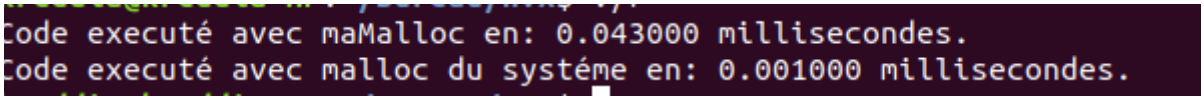
a la fin de la zone critique il suffit de déverrouiller le mutex : pthread_mutex_unlock(&lock),

une foi le travaille de mutex terminer , on peut le détruire

Test de performances:

On a crée un fichier test1.c qui contient un test de la fonction maMalloc et on repris le meme programme mais avec malloc du système et on a calculer le temps d'exécution ,

Les résultats :



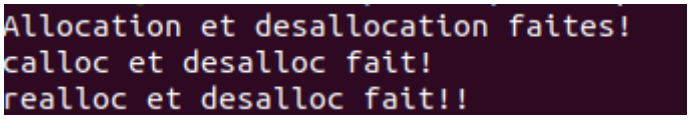
Remarques :

On voit bien que le code exécuté avec notre fonction implémenter « maMalloc » prend plus de temps que la fonction malloc du système .

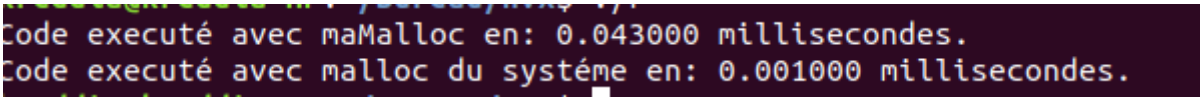
Compilation et exécution :

Pour générer la bibliothèque dynamique (libmalloc.so), on compile le code source (maMalloc.c) avec l'option fPIC et elle se réalise en appellant gcc avec l'option « shared »

- Main.c qui contient le main du fichier maMalloc.c ce dernier contient les implémentations des fonction malloc,calloc,realloc et free,et ainsi on verifie la fonctionnalité de nos implémentations,



- test1.c qui contient le test de performance ,



- threadtest.c qui contient un test des threads appliqué sur le programme maMalloc.c pour les implémentations malloc,calloc ,realloc et free