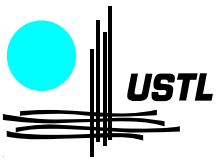

**Licence d'informatique**
**Module de Pratique du C**

Travaux pratiques

# Ma bibliothèque d'allocation mémoire

Philippe MARQUET

Novembre 2005

Ce TP propose d'écrire notre propre version de la bibliothèque d'allocation dynamique de mémoire disponible en C sous Unix. Nous présentons une implantation possible de cette bibliothèque C à l'aide des appels système Unix dans une première partie. Nous envisagerons des améliorations possibles de cette bibliothèque dans une seconde partie. Une troisième partie explique comment utiliser notre propre version de la bibliothèque en remplacement de la bibliothèque standard fourni par le système.

## 1 Bibliothèque standard

### 1.1 Bibliothèque et appels système

La bibliothèque Unix propose un système d'allocation dynamique de mémoire principalement utilisé au travers les fonctions `malloc()` et `free()`. Il existe aussi un appel système d'allocation de mémoire `sbrk()` [1]. Le but de cette section est l'étude de l'écriture des fonctions de la bibliothèque à l'aide de cet appel système.

Détaillons les trois fonctions.

**`void *malloc (unsigned size);`** La fonction `malloc()` de la bibliothèque retourne un pointeur sur un bloc d'au moins `size` octets, qui est correctement aligné selon la contrainte la plus forte de tous les types manipulables en C.

**`void free (void *ptr);`** La fonction `free()` de la bibliothèque libère un bloc précédemment alloué ; `ptr` est un pointeur sur ce bloc.

**`void *sbrk (int incr);`** L'appel système `sbrk()` incrémente l'espace mémoire utilisateur de `incr` octets et retourne un pointeur sur le début de cette zone nouvellement allouée.

Le principe de fonctionnement est le suivant :
- l'utilisateur demande un bloc mémoire via un appel à `malloc()` ;
- `malloc()` demande lui même un bloc mémoire au système via `sbrk()`. Pour éviter le coût de cet appel système, `malloc()` demande un « gros » bloc mémoire à `sbrk()` et n'en utilise qu'une partie pour satisfaire la demande de l'utilisateur. Une structure de données interne à la bibliothèque d'allocation mémoire conserve la zone mémoire obtenue et non utilisée ;
- lors de prochaines demandes de blocs mémoire par l'utilisateur via des appels à `malloc()`, la mémoire excédentaire précédemment obtenue par `sbrk()` est utilisée ;
- les blocs mémoire libérés par `free()` sont gardés dans l'espace mémoire utilisateur et pourront aussi servir à répondre aux demandes suivantes. La mémoire n'est jamais rendue au système avant la fin de l'exécution du programme.

1. On se contentera de l'explication sommaire suivante : un appel système est un appel d'une fonction de bas niveau du système. La notion d'appel système sera détaillée dans le cours de « Programmation des systèmes » au semestre prochain. Par ailleurs, l'emploi de `sbrk()` est maintenant obsolète ; on s'en accommodera dans le cadre de ce TP pour ne pas compliquer les choses.

## 1.2 Possible implémentation de la bibliothèque

L'ensemble des blocs mémoire libres est par exemple gardé par la bibliothèque d'allocation mémoire sous la forme d'une liste chaînée. Chaque bloc contient une taille, un pointeur sur le bloc suivant et l'espace disponible proprement dit. Pour permettre un facile « recollement » de blocs, on conserve par exemple les blocs dans l'ordre croissant des adresses mémoire. De plus, le dernier bloc pointe sur le premier.

On pourra répondre aux questions suivantes en vue de réaliser sa propre implantation de la bibliothèque :

1. Quelle informations doit garder la bibliothèque entre deux appels à `malloc()` ?
2. Quelle(s) information(s) doit-on conserver pour chaque bloc retourné par `malloc()` ? Comment conserver cette information ?

## 1.3 Implémentation de `malloc()`

Lors d'une demande de mémoire, on parcourt la liste des blocs libres jusqu'à rencontrer un bloc suffisamment grand. Cet algorithme est appelé « first-fit » (on choisit le premier qui convient), en opposition à « best-fit » (on choisit le meilleur).

Si le bloc a exactement la taille demandée, on l'enlève de la liste et on le retourne à l'utilisateur. Si le bloc est trop grand, on le divise et on retourne à l'utilisateur un bloc de la taille demandée ; le reste du bloc est gardé dans la liste des blocs libres.

Si la recherche d'un bloc de taille suffisante a échoué, on demande une augmentation de la mémoire utilisateur au système.

## 1.4 Implémentation de `free()`

La libération d'un espace recherche l'emplacement auquel insérer ce bloc dans la liste des blocs libres. Si le bloc libéré est adjacent à un bloc libre, on les fusionne pour former un bloc de plus grande taille. Cela évite une fragmentation de la mémoire et autorise ensuite de retourner des blocs de grande taille sans faire des appels au système.

## 1.5 Optimisation pour des petites allocations

Notre bibliothèque d'allocation dynamique propose aussi une allocation optimisée pour de petits blocs de mémoire. Il est en effet courant d'allouer des petits blocs dont la taille peut être connue à l'avance et pour lesquels il n'est pas nécessaire de mettre en œuvre un mécanisme d'allocation et de libération mémoire aussi lourd que celui présenté ci-dessus.

Nous en présentons ici l'utilisation et une possible implémentation.

### Description de `mallopt()`

On utilise la fonction `mallopt()` pour préciser les caractéristiques des petits blocs à allouer. On passe en paramètre à `mallopt()` une commande et une valeur, paramètre de la commande :

```
int mallopt(int cmd, int val);
```

Les commandes `cmd` sont les suivantes :

**La commande `M_MXFAST`** définit $maxfast$, la taille maximale des blocs qui seront alloués selon l'algorithme d'allocation rapide des petits blocs. Toutes les requêtes à `malloc()` avec une taille inférieure à celle-ci seront optimisées. Initialement, $maxfast$ est nul est l'allocation optimisée n'est pas mise en place.

**La commande `M_NLBLOCKS`** définit $numblks$, le nombre de blocs qui constituent un « tableau de blocs ». C'est-à-dire que si il n'y a plus de petits blocs disponibles lors d'un appel à `malloc()`, la bibliothèque ré-allouera un tel « tableau de blocs ». La valeur par défaut de $numblks$ est 100.

`mallopt()` peut être appelée un nombre quelconque de fois ; mais ne peut plus l'être après l'allocation du premier petit bloc.

**Implémentation de `mallopt`**

Nous proposons ici une implémentation de `mallopt()` pour les deux commandes `M_MXFAST` et `M_NLBLKS`. L'idée est d'allouer, quand nécessaire, un tableau de $numblks$ blocs de taille $maxfast$ et de gérer dans ce tableau un liste chaînées des blocs libres. Initialement, tous les blocs sont chaînés.

Lors d'une demande d'allocation ou de restitution d'un bloc, on met à jour la liste des blocs libres. Il n'est ni nécessaire de maintenir cette liste des blocs libres ordonnées, ni nécessaire de ≪ recoller ≫ la mémoire libérée.

Répondre aux questions suivantes pourra aider à la réalisation de cette implémentation.

1. Quelles informations doivent être conservées entre deux appels à la bibliothèque ? Les appels à `mallopt()` ne font que mettre à jour ces informations.
2. Quelle information doit être associée à chaque tableau de blocs ? Comment garder cette information ? Comment mémoriser l'ensemble des tableaux de blocs ?

# 2 Amélioration de la bibliothèque d'allocation mémoire

Nous allons maintenant proposer des améliorations possibles de notre bibliothèque d'allocation mémoire qui pourront être implémentées dans un second temps.

Le but de ces améliorations est de détecter, autant que faire se peut, les utilisations illégales des segments mémoire alloués dynamiquement par des appels à `malloc()`.

Une telle bibliothèque serait utilisée lors du développement ou de la mise au point d'une application.

## 2.1 Quelles améliorations ?

Nous donnons les points pour lesquels une bibliothèque peut détecter de telles utilisations illégales :

1. Passage à `free()` d'une adresse ne correspondant pas à une adresse précédemment retournée par `malloc()`.
2. Supposition de remplissage des segments mémoire retournés par `malloc()` à zéro.
3. Débordement d'écriture sur un segment alloué par `malloc()`.
4. Utilisation d'un segment après l'avoir rendu par `free()`.
5. Utilisation frauduleuse d'un segment mémoire alloué par `malloc()` dans une des fonctions des bibliothèques de manipulation de mémoire ou de chaînes de caractères (fonctions `str*()` telle `strcpy()`, consultez le manuel `man 3 string` ; `mem*()`, telle `memcpy()`, consultez le manuel `man 3 bstring`).

Une bibliothèque peut aussi intégrer des outils permettant de faciliter la mise au point d'une application en ce qui concerne l'allocation dynamique de mémoire. Par exemple :

6. Vérification du chaînage des blocs gérés par la bibliothèque.
7. Production d'une carte mémoire du tas [2].

2. Le tas désigne les zones mémoires gérées par la bibliothèque d'allocation dynamique.

### 2.2 Quelle implémentation de ces améliorations

Nous reprenons ici chacun des points proposés dans la section précédente et donnons des éléments de solution pour leur mise en place.

1. Afin que `free()()` puisse vérifier qu'une adresse qui lui est passée correspond effectivement à une adresse préalablement retournée par `malloc()`, nous pouvons :
   – garder une liste de ces adresses ; ou
   – lors de l'appel à `malloc()`, allouer un entête devant le segment qui sera retourné et marquer cette entête avec une valeur donnée ; lors du retour du segment à `free()`, on vérifie que la valeur écrite dans le segment est bien celle attendue.
2. Pour détecter les codes qui supposent que les segments mémoire retournés par `malloc()` sont mis à zéro, il nous suffit de remplir ces segments par des valeurs non-nulles ; l'exécution du code devrait impliquer une erreur qui nous espérons pourra être découverte par le programmeur.
3. Pour détecter les débordements d'écriture sur un segment alloué par `malloc()`, une solution consiste à allouer des segments de taille plus grande que demandée et de remplir cet espace par des valeurs arbitraires. Lors de la remise du segment à `free()`, on vérifie que cet emplacement n'a pas été réécrit.
4. Pour éviter que le programmeur n'utilise encore un segment après l'avoir rendu par `free()`, nous pouvons réécrire par des valeurs arbitraires tous les segments rendus.
5. Pour éviter les utilisations frauduleuses des segments mémoire alloués par `malloc()` dans une des fonctions des bibliothèques de manipulation de mémoire ou de chaînes de caractères, nous pouvons surcharger toutes ces fonctions par nos propres versions. Ces versions peuvent regarder si les paramètres passés référencent des segments alloués dynamiquement, et si c'est le cas qu'aucune lecture ou écriture ne dépasse la longueur du segment.
6. Lors de chaque appel à notre bibliothèque ou par des appels spécifiques à une fonction supplémentaire, nous pouvons vérifier l'intégrité du chaînage des blocs et donc vérifier que des écritures intempestives n'auraient pas détruites des informations.
7. Une autre fonction de notre bibliothèque pourrait produire un affichage de l'ensemble des informations concernant les blocs alloués par `malloc()`.

## 3 Validation de la bibliothèque d'allocation mémoire

Nous présentons ici le moyen de valider notre bibliothèque d'allocation mémoire.

Cette bibliothèque fournie des fonctions `malloc()` et `free()` qui existent déjà dans la bibliothèque standard. L'idée est de faire en sorte que les programmes existant qui utilisent les fonctions `malloc()` et `free()` utilisent nos versions de ces fonctions.

Nous allons réaliser cela en deux temps :
– dans un premier temps nous considérerons les programmes dont nous disposons des codes sources (section 3.2) ;
– dans un second temps nous considérerons les programmes dont ne disposons pas des codes sources (section 3.3).

### 3.1 Bibliothèque de remplacement

Nous nous proposons de développer une bibliothèque de remplacement de la bibliothèque d'allocation dynamique de mémoire. L'interface de notre bibliothèque doit donc être la même que celle de la bibliothèque standard. Pour que les appels aux fonctions d'allocation mémoire restent cohérents, il faut que notre bibliothèque remplace *toutes* les fonctions de la bibliothèque standard. Il s'agit des quatre fonctions

```
void * calloc (size_t nmemb, size_t size);
void * malloc (size_t size);
void free (void * ptr);
void * realloc (void * ptr, size_t size);
```

Vous consulterez la documentation en ligne pour une description des fonctions annexes `calloc()` et `realloc()`.

## 3.2 Surcharge des fonctions standard

Supposons que nous produisions une bibliothèque `bam` (*bam* pour bibliothèque d'allocation mémoire) contenant notre implantation des fonctions d'allocation mémoire. Le fichier `bam.c` produit donc `bam.o`.

Soit un programme `essai.c` existant (ou que nous créons pour le besoin de tester notre bibliothèque) qui utilise les fonctions `malloc()` et `free()`.

Sans toucher au fichier source, nous pouvons utiliser notre bibliothèque en produisant un exécutable de la manière suivante :

```
gcc -Wall -Werror -ansi -pedantic -o essai essai.o bam.o
```

Cependant, il peut être intéressant pour notre bibliothèque de passer des paramètres supplémentaires aux fonctions, tels le nom du fichier source et le numéro de la ligne dans lequel figure l'appel. Les messages d'erreur pourront alors afficher ces informations et ainsi aider le programmeur à rechercher ses erreurs. Le prototype de notre fonction d'allocation mémoire sera alors différent de celui de `malloc()` :

```
/* malloc() replacement */
void *
bam_malloc (size_t size,        /* as in malloc() */
            char *filename,     /* source filename of the function call */
            unsigned line);     /* source line number of the f. call */
```

Soit un extrait de source du programme `essai.c` :

```
char * str = malloc(1024);
```

Nous voulons appeler notre fonction d'allocation `bam_malloc()` au lieu de `malloc()`. Plutôt que de modifier le code source de `essai.c`, nous allons utiliser le préprocesseur pour ce faire, par exemple avec une macro-définition telle celle-ci :

```
#define malloc(sz)      bam_malloc(sz, __FILE__, __LINE__)
```

Les deux macros `__FILE__` et `__LINE__` sont définies par le compilateur et désignent respectivement une chaîne de caractères contenant le nom du fichier source, et le numéro de la ligne courante dans ce fichier source.

Il nous suffit pour utiliser notre bibliothèque dont l'interface des fonctions est différentes de placer les macro-définitions dans un fichier d'entête `bam.h` et d'ajouter la ligne

```
#include "bam.h"
```

dans le fichier source `essai.c`.

## 3.3 Édition de liens dynamique

Par ailleurs, il est aussi intéressant de pouvoir récupérer les appels à la bibliothèque standard qui sont faits dans du code pour lequel nous n'avons pas accès au source ; par exemple dans les exécutables des commandes du système ou dans les fonctions des bibliothèques fournies avec le système.

Pour ces appels, notre bibliothèque ne peut avoir accès aux informations telles le nom du fichier source ou le numéro de la ligne et ne pourra donc pas les utiliser. Dans ce cas, nous allons coder les fonctions `malloc()` de la manière suivante dans un fichier `malloc.c` :

```
/* Redefinition of malloc () */
void *
malloc (size_t size)
{
    /* unknown source filename and line */
    return bam_malloc(size, (char *)0, 0);
}
```

Lors de la production d'un exécutable, par exemple celui de la commande `ls`, le système ne réalise par l'édition de liens avec la bibliothèque standard. C'est au moment de l'exécution de la commande qu'une édition de liens *dynamique* est réalisée. Ainsi il est possible de mettre à jour la bibliothèque système sans avoir à recompiler l'ensemble des commandes.

Nous allons profiter de ce mécanisme d'édition de liens dynamique pour remplacer les fonctions d'allocation dynamique par nos propres versions.

Dans un premier temps il nous faut produire une bibliothèque dynamique (d'extension `.so`) contenant tout ce qui est nécessaire à nos fonctions `malloc()` et `free()`. On produit tout d'abord les `.o` nécessaires, `malloc.o` et `bam.o` :

```
gcc -Wall -Werror -ansi -pedantic -D_XOPEN_SOURCE=500 -g -c malloc.c
gcc -Wall -Werror -ansi -pedantic -D_XOPEN_SOURCE=500 -g -c bam.c
```

On produit ensuite le fichier `libmalloc.o` de la manière suivante :

```
gcc -shared -Wl,-soname,libmalloc.so -o libmalloc.so malloc.o bam.o
```

Dans un second temps, on précise que l'on désire charger cette bibliothèque au début de l'édition de liens dynamique en positionnant la variable shell `$LD_PRELOAD` :

```
% setenv LD_PRELOAD ./libmalloc.so
```

ensuite, toute exécution de commande utilisera cette bibliothèque qui masquera les fonctions de la bibliothèque standard. L'exemple fourni (qui affiche une trace des appels à `malloc()` en fin d'exécution, voir la section 3.4) produit l'exécution suivante :

```
% ls
libmalloc.so*  malloc.o       phm_malloc.h  sc_malloc*   tools.h
malloc.c       phm_malloc.c   phm_malloc.o  sc_malloc.c
malloc.make    phm_malloc.c~  sc-malloc.c   sc_malloc.o
% setenv LD_PRELOAD ./libmalloc.so
% ls
libmalloc.so*  malloc.o       phm_malloc.h  sc_malloc*   tools.h
malloc.c       phm_malloc.c   phm_malloc.o  sc_malloc.c
malloc.make    phm_malloc.c~  sc-malloc.c   sc_malloc.o
------------------------------
 Trace des appels a malloc()
-------------------------------------------------------------
181 appel(s) a malloc()
Dans ??,        ligne 0,        352 octets demandes
Dans ??,        ligne 0,        appel de free()
Dans ??,        ligne 0,        42 octets demandes
Dans ??,        ligne 0,        appel de free()
Dans ??,        ligne 0,        42 octets demandes
Dans ??,        ligne 0,        28 octets demandes
Dans ??,        ligne 0,        39 octets demandes
Dans ??,        ligne 0,        24 octets demandes
Dans ??,        ligne 0,        100 octets demandes
Dans ??,        ligne 0,        6 octets demandes
[...]
```

Pour désactiver ce mécanisme, il nous suffit de supprimer la définition de la variable shell `$LD_PRELOAD` :

```
% unsetenv LD_PRELOAD
```

### 3.4 Exemple

Afin de donner quelques éléments concrets de l'utilisation d'une bibliothèque d'allocation en remplacement de la bibliothèque standard, un exemple complet est disponible en ligne. Cet exemple rudimentaire se contente de compter les appels à `malloc()` et d'identifier les noms des fichiers et les numéros de lignes de ces appels. Se référer au répertoire `/home/enseign/PDC/src/malloc/`.

Exécutez une à une les commandes suivantes, observez les résultats et tâchez de comprendre le fonctionnement de cet exemple :

```
% make sc_malloc
% ./sc_malloc
% make libmalloc.so
% ls
% setenv LD_PRELOAD ./libmalloc.so
% ls
% grep foo *.c
```

## 4 Travail à rendre

Il s'agit donc d'écrire

1. Votre propre version des fonctions `malloc()` et `free()` (sous forme de macro appelant les fonctions `bam*()` et sous forme de remplacement des fonctions `malloc()` et `free()` standard).
2. Une implantation de `mallopt()`.
3. Votre propre version des fonctions `calloc()` et `realloc()`.
4. Éventuellement, vos propres versions des fonctions de manipulation de chaînes de caractères (`man 3 string`) et de manipulation de mémoire (`man 3 bstring`).
5. Des fonctions de vérification de la cohérence du tas et d'affichage de celui-ci.
6. Des exemples bien choisis de programmes utilisant votre bibliothèque.

Il est vraisemblable que vous n'implanterez pas les l'ensemble des propositions d'amélioration suggérées dans le sujet. Identifiez clairement dans votre compte-rendu de TP ce que vous avez implanté.