Erick Gallesio, Stéphane Lavirotte, Jean-Paul Rigault, Hélène Collavizza
Polytech'Nice – Sophia
SI3
2018-2019

TD n°8
Gestion Mémoire

1 Objectif

Le but de cet exercice est de réaliser un allocateur dynamique de mémoire, c'est- à-dire un substitut aux fonctions `malloc` et `free` de C.

Les allocateurs de mémoire généraux sont parmi les programmes système les plus délicats à réaliser et à tester, mais aussi ceux qui peuvent avoir une influence considérable sur les performances en temps et en mémoire. Nous ne prétendons pas réaliser ici un allocateur très sophistiqué, seulement donner une idée des problèmes.

Cet exercice est aussi l'occasion de manipuler à un niveau fin les pointeurs de C, en mettant vraiment les mains dans le cambouis, comme on a souvent à le faire en programmation système. L'exercice n'est pas facile, même si le code en est court. Lisez bien les spécifications et les remarques qui suivent. Certains choix de conception ne deviendront clairs qu'après avoir codé une solution.

Comme d'habitude, pour vous éviter de perdre trop de temps pour la mise en route, vous devrez récupérer l'archive contenant des pièces du puzzle :

http://trolen.polytech.unice.fr/cours/progsys/td08/td08_distrib.zip

Commencez par lire entièrement le sujet avant de vous lancer à programmer.

2 Présentation du problème

2.1 Les fonctions `malloc` et `free` d'ANSI C

En C, la fonction `malloc` permet au programmeur d'allouer dynamiquement de la mémoire, et la fonction `free` lui permet de rendre cette mémoire afin de la recycler pour l'utiliser dans un éventuel `malloc` suivant.

Voici un exemple d'utilisation :

```
struct Data { // Une structure de données
    char nom[100];
    int age;
};
...
// On alloue dynamiquement un objet de ce type
// La fonction malloc retourne un pointeur sur la zone allouée
struct Data *pdata = malloc(sizeof(struct Data));
// On peut maintenant utiliser librement cet objet
pdata->age = 12;
strcpy(pdata->nom, "Peter Pan");
...
// Et quand on n'en a plus besoin le liberer
free(pdata);
// Attention : ici le pointeur pdata n'est plus valide !
```

Cependant, la plupart des systèmes d'exploitation ne réalisent pas de manière primitive cette gestion du recyclage. Les fonctions `malloc` et `free` ne sont donc pas, en général, des appels systèmes mais des fonctions de bibliothèque.

On peut d'ailleurs se demander pourquoi des fonctions aussi importantes ne sont pas directement réalisées par le système d'exploitation. La raison en est très simple : il est très difficile d'écrire un allocateur général de mémoire dynamique, qui doit être à l'aise aussi bien pour allouer un grand nombre de petits objets qu'un grand nombre de très grands ou encore un mélange des deux. Des compromis de conception sont indispensables et les mauvais choix peuvent entraîner des pertes de performances parfois considérables. Donc il est préférable de ne pas figer les algorithmes de gestion mémoire dans le noyau. En les réalisant sous forme de fonctions de bibliothèque, on peut les

changer et les remplacer facilement pour, par exemple, les adapter à un schéma d'utilisation mémoire particulier, pour lequel on peut imaginer des algorithmes plus efficaces que les compromis généraux.

2.2 La fonction UNIX `sbrk`

Si le système d'exploitation ne réalise pas lui-même la gestion du recyclage, il doit cependant collaborer un peu pour permettre la réalisation de la fonction `malloc`. Le minimum qu'il ait à faire est de permettre d'augmenter l'espace mémoire d'un programme. Sous UNIX (et donc GNU/Linux), ceci est réalisable par l'appel-système sbrk[1] (`man sbrk`, donc…).

Cette primitive s'utilise très simplement : il suffit de faire

```
void *pnew = sbrk(incr);
```

où `incr` est un entier non signé, pour que le segment de données du programme s'accroisse de (au moins) `incr` octets. La valeur de retour est un pointeur sur le début de la zone supplémentaire ainsi allouée. Notez bien que cette zone n'a absolument aucune structure ; c'est juste des octets à la suite les uns des autres, et c'est aux fonctions `malloc` et `free` qu'il appartiendra de la structurer.

Si le système ne peut plus allouer de mémoire supplémentaire, `sbrk` retourne `-1`, ce qui n'est pas une très bonne idée car `-1` n'est pas une valeur de pointeur (!) et cela rend le test un peu pénible :

```
if (pnew == ((void *) -1))
    fprintf(stderr, "Plus de memoire\n");
```

3 Un allocateur dynamique simple

3.1 Spécification de l'interface

Bien que nous ayons annoncé que c'était très difficile, nous allons réaliser une version simple de `malloc` et `free`. Évidemment notre version ne sera pas aussi évoluée ni aussi efficace que celles que l'on trouve dans les systèmes modernes. Mais elle sera complète et permettra de mettre en évidence les difficultés de la tâche.

Pour ne pas les confondre avec les versions standard, nous nommerons nos fonctions `mymalloc` et `myfree`. Leurs prototypes seront analogues à ceux du standard :

```
void *mymalloc(size_t size);
void myfree(void *p);
```

La fonction `mymalloc` retourne un pointeur sur un bloc assez grand pour contenir un objet de taille `size` caractères (size est un entier long non signé). En cas d'échec, `mymalloc` retourne le pointeur `NULL`.

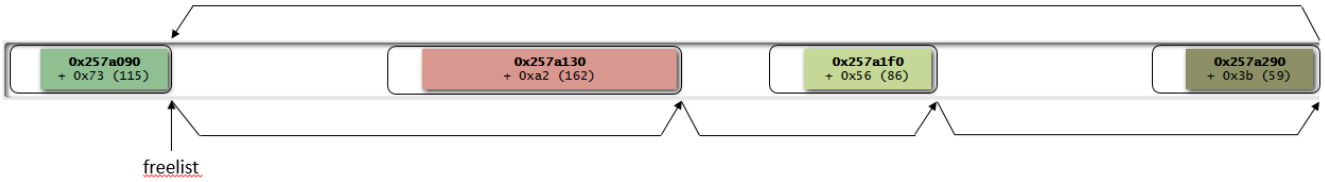
Quant à `myfree`, elle libère la zone pointée par `p` afin qu'elle soit réutilisable par un futur `mymalloc` dans le même programme ; après cet appel `p` est invalide (mais pas nul! en fait sa valeur n'est pas modifiée). Bien entendu, pour pouvoir appeler `myfree`, `p` doit avoir une valeur qui est le résultat d'un précédent `mymalloc`.

3.2 Mise en œuvre

Dans notre implémentation, la fonction `malloc` utilise donc la fonction système `sbrk` qui permet de demander au système de changer la taille de la zone mémoire allouée à un processus. Lorsque la fonction `free` est appelée l'espace qui était alloué à l'adresse donnée est remis dans une liste de blocs libres. Cette liste pourra être gérée comme une

[1] Les fonctions `malloc` et `free` font partie de la norme ANSI C et donc de POSIX. Ce n'est pas le cas de `sbrk` qui est spécifique à UNIX : d'autres systèmes d'exploitation peuvent proposer un mécanisme fondamentalement différent pour obtenir de la mémoire du système.

liste circulaire. La situation que l'on a dans le cas général (c'est-à-dire après plusieurs appels à `malloc` et `free`) est représentée ci-dessous.



Les zones blanches correspondent aux blocs qui ont déjà été libérés par un appel à `free` ; les zones en couleur, quant à elles, correspondent aux régions allouées par `malloc` qui sont encore utilisées.

3.2.1 Allocation mémoire

Lorsque l'utilisateur demande *n* octets de mémoire, l'algorithme d'allocation parcourt la liste des blocs libres à la recherche d'un bloc assez grand pour satisfaire la demande. Si un tel bloc existe, les *n* premiers octets de ce bloc sont retournés à l'utilisateur, tandis que le reste du bloc est laissé dans la liste des bloc libres. Dans le cas contraire, la fonction `sbrk` est appelée pour faire grossir le segment de données du processus.

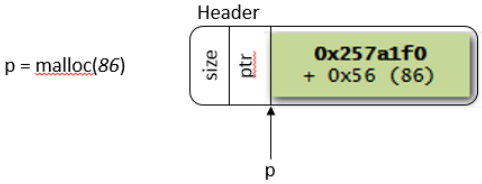
Le choix du « *meilleur bloc* » parmi les blocs libres peut se faire suivant la stratégie « *best fit* », « *first fit* », « *worst fit* », etc.

3.2.2 Libération mémoire

Lorsque la mémoire est rendue au système, on se contente de remettre le bloc dans la liste des blocs libres. Bien sûr, si ce bloc est contigu avec d'anciens blocs libres, il sera fusionné avec ceux-ci.

3.2.3 Entête de bloc

Afin de gérer les blocs libres, les fonctions `malloc` et `free` ont besoin, pour chaque bloc, d'un pointeur de chaînage et d'un entier indiquant la taille utile du bloc. Par conséquent, chaque zone mémoire retournée à l'utilisateur sera précédée par un entête contenant ces informations :



Un entête de bloc pourra être représenté par le type suivant :

```
typedef struct header {     /* Header de bloc */
  unsigned int size;        /* Taille du bloc */
  struct header *ptr;       /* Bloc libre suivant */
} Header;
```

3.3 Problèmes d'alignement

Généralement, les processeurs imposent certaines contraintes sur les adresses auxquelles les données peuvent être stockées (par exemple, un double doit être rangé à une adresse multiple de 8). En C, la fonction `malloc`, doit s'occuper des problèmes d'alignement et rendre une adresse à laquelle on peut ranger des objets de type quelconque. Pour cela, on supposera que le type le plus contraignant est dénoté par la constante `MOST_RESTRICTING_TYPE` (sur un PC ce type pourra être défini à double). Par conséquent, le type Header sera redéfini de la façon suivante :

```
#define MOST_RESTRICTING_TYPE double // Pour s'aligner sur des frontières multiples
                                     // de la taille du type le plus contraignant
```

TD n°8
Gestion Mémoire

```
typedef union header {              // Header de bloc
  struct {
    unsigned int size;              // Taille du bloc
    union header *ptr;              // bloc libre suivant
  } info;
  MOST_RESTRICTING_TYPE dummy;      // Ne sert qu'a provoquer un alignement
} Header;
```

4 Travail demandé

Le travail demandé pour ce TD consiste tout d'abord à réécrire les fonctions `malloc` et `free` (puis dans un second temps `calloc` et `realloc`) de la bibliothèque standard de C.

4.1 Implémentation de mymalloc et myfree

Votre fonction `mymalloc` suivra une stratégie « *first fit* » pour choisir le bloc qui sera retenu dans la liste des blocs libres. Cette stratégie consiste à choisir le premier bloc de taille suffisante dans la liste de bloc libres de le couper en deux[2] et de laisser la partie inutilisée dans la liste des blocs libres. Cette stratégie n'est pas optimale, puisqu'elle va morceler la mémoire, mais elle a l'avantage d'être simple à implémenter.

Voici un algorithme en pseudo code simplifié pour vous faciliter l'implémentation de cette fonction :

```
je parcours la liste
   si je suis sur un bloc de taille >= à la taille souhaitée
        si je suis sur un bloc de taille = à la taille cherchée
              supprimer le bloc de la liste
              retourner le pointeur comme résultat de la fonction
        sinon
              découper le zone et ajouter la zone restante libre à la liste
              retourner le pointeur comme résultat de la fonction

   si j'ai réalisé un tour de liste (donc sans trouver)
        allouer une nouvelle zone à la liste (obtenue par sbrk)
        ajouter la zone à la liste (via un appel à free)
```

Pour tester votre fonction `myfree`, vous pouvez essayer de libérer tous les blocs que vous avez alloués. Normalement, votre algorithme doit regrouper tous les blocs contigus et vous devriez donc avoir tous vos blocs regroupés (après libération de tous les blocs alloués, bien entendu).

Voici un algorithme en pseudo code simplifié pour vous faciliter l'implémentation de cette fonction :

```
parcourir la liste jusqu'à « la bonne place »
fusionner avec la zone suivante si nécessaire sinon mettre à jour la liste
fusionner avec la zone précédente si nécessaire sinon mettre à jour la liste
```

Un programme de test simple vous est fourni dans le fichier `test-malloc.c`.

4.2 Visualisation de l'état du tas (heap)

Il est toujours difficile de se représenter l'état de la mémoire et en particulier de la mémoire allouée dynamiquement. Afin de vous faciliter la tâche, nous vous proposons d'utiliser un petit script Python[3] permettant de générer un historique de l'état du tas (*heap* en anglais), après l'appel de chaque instruction allouant ou récupérant de la mémoire).

[2] sauf s'il fait juste la bonne taille bien sûr.
[3] https://github.com/wapiflapi/villoc

Ce script a été modifié pour ajouter la possibilité de visualiser les opérations de type `sbrk` et de prendre en compte vos fonctions `mymalloc` et `myfree`. Pour générer ce type de graphique, il faudra récupérer les différents appels aux fonctions `mymalloc` et `myfree` ainsi qu'à l'appel système `sbrk`. Nous utiliserons la fonction `ltrace` (vue lors du premier TD) permettant de tracer les appels à des librairies externes, mais aussi aux appels systèmes avec l'option `-S`. Le résultat de cette commande sera envoyé en tant qu'entrée du script Python générant une page html avec la frise chronologique de l'état du tas.

```
ltrace -S ./test-malloc.exe |& villoc/villoc.py --raw - malloc.html
```

Pour vous faciliter cette visualisation, nous avons créé une entrée `test` dans le `Makefile`. Donc il suffira de lancer la commande suivante pour générer le fichier html contenant une représentation graphique de la mémoire :

```
make test
```

4.3 Implémentation de `mycalloc` et `myrealloc`

Quant aux fonctions `mycalloc` et `myrealloc`, elles sont simples à écrire et s'expriment en fonction de `mymalloc` et `myfree`. C'est une extension possible et simple de votre travail pour le compléter.