Vincent Simonet (http://cristal.inria.fr/~simonet/)

Allocation mémoire

D'après l'épreuve du concours *Centrale–Supélec* 1999

On examine dans ce sujet le problème de l'allocation mémoire. On peut voir la mémoire centrale d'un ordinateur comme un tableau (ou vecteur) m dont les éléments (qui représentent l'unité élémentaire de mémoire) sont appelés mots mémoire *(ou* mots*). La taille n du tableau est généralement une puissance de 2, soit $n = 2^N$. Les indices de ce tableau (qui sont des entiers compris entre 0 et $n - 1$) sont appelés* adresses. *Un mot est de taille suffisante pour stocker une adresse.*

1 Gestion de blocs libres par liste

Tout programme exécuté sur l'ordinateur effectue des requêtes mémoire : soit en demandant l'allocation d'un bloc mémoire (une suite contigüe de mots) de taille donnée, soit en libérant un bloc mémoire. Ces allocations et libérations sont effectuées en très grand nombre, et de manière totalement imprévisible, car elles diffèrent à chaque exécution du programme, en fonction des données fournies.

Deux problèmes se posent immédiatement :
– trouver une structure de données adéquate pour représenter les blocs libres,
– concevoir un algorithme pour trouver un bloc libre de taille n et le réserver.

On peut répondre à la première question en créant une liste pour représenter les blocs libres (on verra par la suite d'autres possibilités). En Caml, on utilisera une liste, chaque élément contenant la taille du bloc libre en nombre de mots et l'adresse de ce bloc. Elle peut être triée (par adresses ou tailles) ou non. Initialement, la liste f (pour *free*) contient un seul bloc de taille 2^N et d'adresse 0.
Les fonctions d'allocation et de libération devront modifier cette liste.

1.1 Allocation

▶ **Question 1** *Écrivez une fonction* alloc_first *qui alloue un bloc en utilisant le premier bloc dans la liste dont la taille est supérieure à la taille demandée. La fonction prendra comme arguments la taille du bloc demandé et la liste des blocs libres. Elle rendra une paire formée de l'adresse du bloc alloué et de la nouvelle liste des blocs libres.*

```
alloc_first: int -> (int * int) list -> int * (int * int) list
```

▶ **Question 2** *Écrivez ensuite* alloc_best *qui utilise un bloc dont la taille se rapproche le plus de la taille demandée.*

On se donne la situation de départ et les deux requêtes suivantes : deux blocs initialement libres de taille 768 et 512 et trois requêtes consécutives : 500, 400 et 300.

▶ **Question 3** *Que donne chacune des deux méthodes ? Laquelle est la plus intéressante ? Montrer par un exemple simple que l'on peut trouver une situation inverse.*

1.2 Libération

On considère maintenant le problème de la libération d'un bloc alloué : il s'agit de remettre ce bloc dans la liste f, en le fusionnant le cas échéant avec d'éventuels blocs adjacents. On suppose que la liste des blocs libres est triée par adresse croissante, et que l'algorithme d'allocation est celui donné par la fonction alloc_first.

▶ **Question 4** *Écrivez la fonction* free *qui va libérer un bloc de mémoire et l'ajouter dans la liste des blocs libres. Vous prendrez garde à fusionner le bloc libéré avec d'éventuels blocs adjacents.*

2 Gestion des blocs libres par liste chaînée dans la mémoire

2.1 Liste simplement chaînée

En réalité, la mémoire nécessaire pour gérer la liste f est également située dans la mémoire de l'ordinateur. Au lieu de créer une structure annexe pour la gestion des blocs libres, on stocke l'information nécessaire à cette gestion dans *les blocs eux-mêmes*. Ainsi, pour allouer (ou spécifier) un bloc libre de taille utile s, on prendra un bloc de taille $s + c$, c étant une constante entière petite devant la taille des blocs réclamés (ce qui ne posera aucun problème), les mots supplémentaires servant à stocker de l'information.
Dans les questions suivantes, on prendra $c = 2$. Si a est l'adresse d'un bloc de mémoire libre, le mot situé à l'adresse a contiendra la taille s du bloc libre et le mot situé à l'adresse $a + 1$ contiendra l'adresse du bloc libre suivant, le bloc proprement dit étant formé des s mots suivants. Un bloc libre situé à l'adresse a a donc la forme suivante :

a	s (taille du bloc)
$a+1$	a' (bloc libre suivant)
$a+2$ $\vdots$ $a+2+s$	(espace libre)

Nous supposerons que le dernier bloc libre de la liste contient, comme adresse du bloc suivant, l'entier n (la taille de la mémoire).

▶ **Question 5** *Modifiez les fonctions* alloc_first *et* free *pour ne plus utiliser de structure de liste annexe, mais travailler directement dans le tableau m.*

```
alloc_first': int -> int vect -> int
free': int -> int -> int vect
```

Après une utilisation répétée de ces deux foncions avec une situation de départ où toute la mémoire est libre, une nette tendance se dégage : les blocs de petite taille s'accumulent au début de la liste des blocs libres, si bien qu'il faut chercher assez loin dans la liste dès que l'on veut allouer un bloc de taille s pour s suffisamment grand.

▶ **Question 6** *Suggérez une modification simple de l'algorithme pour la fonction* alloc_first ' *de telle sorte que les blocs de petite taille ne s'accumulent pas en un point particulier de la liste et la liste des blocs libres reste triée par adresse croissante dans le but d'utiliser la fonction* free*.*

2.2 Liste doublement chaînée

On réserve maintenant un peu de place supplémentaire pour ajouter les informations suivantes :

– Le premier mot d'un bloc occupé contient l'*opposé* de sa taille (supposée non nulle). Un bloc occupé à l'adresse a a donc la forme suivante :

a	$-s$(taille du bloc)
$a+1$ $\vdots$ $a+1+s$	(données)

– Chaque bloc libre contient également l'adresse du bloc libre précédent. Un bloc libre à l'adresse a a donc la forme suivante :

a	s(taille du bloc)
$a+1$	a'' (bloc libre précédent)
$a+2$	a' (bloc libre suivant)
$a+3$ $\vdots$ $a+3+s$	(espace libre)

▶ **Question 7** *Modifiez les fonctions* alloc_first *et* free *pour utiliser cette nouvelle structure de données de manière à améliorer l'efficacité de la libération.*

▶ **MP & MP* – Option Informatique**
Année 2002–2003, Deuxième TP Caml
Vincent Simonet (http://cristal.inria.fr/~simonet/)

Allocation mémoire

Un corrigé

▶ Question 1

```
let rec alloc_first s = function
    [] -> failwith "Out_of_memory"
  | (a', s') :: tail when s' > s ->
      a', (a' + s, s' - s) :: tail
  | (a', s') :: tail when s' = s ->
      a', tail
  | (a', s') :: tail ->
      let a, tail' = alloc_first s tail in
      a, (a', s') :: tail'
;;
```

▶ Question 2

```
let rec find_best s = function
    [] -> max_int
  | (_, s') :: tail ->
      if s' < s then find_best s tail
      else min s' (find_best s tail)
;;

let alloc_best s f =
  let best = find_best s f in
  let rec aux = function
      [] -> failwith "Out_of_memory"
    | (a', s') :: tail when s' = best ->
        if s' = s then (a', tail)
        else (a', (a' + s, s' - s) :: tail)
    | (a', s') :: tail ->
        let a, tail' = aux tail in
        a, (a', s') :: tail'
  in
  aux f
;;
```

▶ Question 4

```
let rec free a s = function
    [] -> [a, s]
  | [a1, s1] when a1 + s1 = a -> [a1, s1 + s]
  | (a1, s1) :: tail when a < a1 ->
      if a + s = a1 then (a, s + s1) :: tail
      else (a, s) :: (a1, s1) :: tail
  | (a1, s1) :: (a2, s2) :: tail ->
      begin match a1 + s1 = a, a + s = a2 with
        false, false -> (a1, s1) :: (a, s) :: (a2, s2) :: tail
      | true, false -> (a, s1 + s) :: (a2, s2) :: tail
      | false, true -> (a1, s1) :: (a, s + s2) :: tail
      | true, true -> (a1, s1 + s + s2) :: tail
      end
  | (a1, s1) :: tail ->
      (a1, s1) :: (free a s tail)
;;
```

▶ Question 5

```
let alloc_first' s m =
  let n = vect_length m in
  let rec aux a =
    if a >= n then failwith "Out_of_memory";
    if m.(a) < s then aux m.(a + 1)
    else begin
      m.(a) <- m.(a) - s;
      a + m.(a) + 2
    end
  in
  aux 0
;;

let try_merge a m =
  let n = vect_length m in
  if m.(a + 1) < n && a + m.(a) + 2 = m.(a + 1) then begin
    m.(a) <- m.(a) + m.(m.(a + 1)) + 2;
    m.(a + 1) <- m.(m.(a + 1) + 1)
  end
;;

let free a s m =
  let rec aux a' =
    if a > m.(a' + 1) then aux m.(a' + 1)
    else begin
      m.(a) <- s - 2;
      m.(a + 1) <- m.(a' + 1);
      m.(a' + 1) <- a;
      try_merge a m;
      try_merge a' m
    end
  in
  aux 0
;;
```