Sujet : Allocation mémoire

Le jury attend une présentation de 35 minutes s'appuyant sur ce sujet, en forme de cours, pédagogique, structurée et évitant la paraphrase. Le texte se conclut par des pistes de réflexion facultatives dont vous pouvez vous saisir ; au-delà de ces pistes proposées, toute initiative personnelle pertinente est appréciée. Vous n'êtes pas obligé de traiter le texte dans son intégralité, mais si seule une partie est traitée elle doit l'être de manière particulièrement approfondie.

L'exposé doit intégrer une (ou plusieurs) illustrations informatiques. Il doit également contenir une discussion autour d'une dimension éthique, sociétale, environnementale, économique ou juridique en lien avec le texte ; une des pistes de réflexion est spécifiquement conçue à cet effet.

1 Résumé

On présente les problèmes liés à la gestion dynamique de la mémoire utilisée par un programme en cours d'exécution : allocation de mémoire, récupération et réutilisation automatique de zones mémoire qui ne sont plus utilisées.

Mots clés: structures de données, graphes, gestion mémoire

2 Organisation de la mémoire

Lors de son exécution, un programme a besoin de mémoire vive pour stocker les données sur lesquelles il opère. La mémoire est mise à la disposition du programme par l'environnement d'exécution, et est divisée en trois catégories.

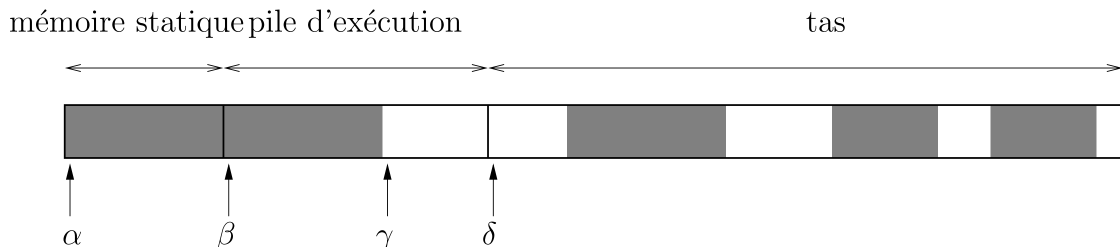
- La mémoire attribuée aux données globales de taille fixe : celle-ci peut être définie et dimensionnée lors de la compilation du programme. Cette partie de la mémoire est appelée *mémoire statique.*

- la mémoire attribuée aux données de taille fixe locales aux procédures et fonctions : la quantité de mémoire nécessaire est déterminée à la compilation pour chaque procédure et pour chaque fonction, mais la quantité totale de mémoire nécessaire pour stocker les variables locales de toutes les fonctions en cours d'exécution ne peut être calculée à la compilation, car un nombre indéterminé de fonctions et de procédures peuvent être actives à un instant donné. Cette partie de la mémoire est appelée *pile d'exécution.*

- la mémoire attribuée aux données locales et globales dont la taille n'est pas déterminable à la compilation : elle peut être fonction des entrées fournies au programme, ou elle peut varier au cours de l'exécution du programme. Cette partie de la mémoire est appelée *tas*.

Nous nous intéressons dans ce texte au fonctionnement du *gestionnaire mémoire* qui est intégré soit au programme, soit à l'environnement d'exécution, et qui attribue au programme les zones mémoire dont celui-ci a besoin, tant dans la pile que dans le tas. Nous supposerons que la mémoire mise à la disposition du programme est un tableau linéaire `MEM` de taille fixe. Cette taille est fonction de l'ordinateur qui exécute le programme, et n'est connue qu'à l'exécution. Une cellule du tableau `MEM` est appelée *mot*. Il s'agit d'un nombre fixé A de bits dépendant de l'architecture de l'ordinateur qui permettent de coder soit une donnée scalaire (nombre entier, caractère, ...), soit un indice valide pour le tableau `MEM`. On supposera par la suite qu'on a affaire à une architecture de 64 bits ($A = 64$). Les données scalaires sont appelées *atomes* et les indices pour le tableau `MEM` sont appelés *pointeurs* ou *adresses*.

La figure ci-dessous montre un exemple d'organisation de la mémoire attribuée à un programme à un instant de son exécution. Les parties de `MEM` effectivement utilisées sont grisées, les parties libres à cet instant sont blanches. $\alpha, \ldots, \varepsilon$ sont des pointeurs. α indique le début de la mémoire statique. β indique le début de la pile d'exécution, c'est-à-dire le premier mot de `MEM` attribué à la pile ($\beta - 1$ est donc l'adresse du dernier mot de la mémoire statique). γ indique le début de la partie libre de la pile, c'est-à-dire le premier mot disponible pour empiler une nouvelle donnée. δ indique le début de la partie réservée au tas et ε le premier mot de `MEM` non utilisable.



La gestion de la pile est simple: pour empiler une nouvelle donnée tenant sur un mot il suffit de vérifier que $\gamma < \delta$, de stocker la donnée en `MEM`$[\gamma]$ et d'incrémenter γ. Pour dépiler la dernière donnée empilée, on vérifie que $\gamma > \beta$, on décrémente γ et on renvoie la donnée stockée en `MEM`$[\gamma]$. Le cas de données occupant plusieurs mots consécutifs se traite de façon analogue.

La partie réservée au tas est découpée en zones mémoires. Ces zones peuvent évoluer au cours de l'exécution du programme mais d'une part chacune de ces zones doit toujours être un bloc de mots consécutifs et d'autre part l'ensemble des zones réalise une partition du tas, autrement dit toute case mémoire du tas appartient à une et une seule zone. La gestion du tas est plus complexe que celle de la pile car les zones libres ne sont pas nécessairement contiguës.

On dit qu'une zone a pour *adresse* a et, pour *taille totale* $n + 1$ et pour *taille utile* n si elle est constituée de $n+1$ mots d'adresses $a-1$, a, ..., $a+n-1$. Le mot d'adresse $a - 1$ est alors appelé *mot d'en-tête* de la zone et les n mots d'adresses a,

$\ldots, a+n-1$ sont appelés *les mots utiles* ou *la partie utile* de la zone. *L'adresse d'une zone est donc l'indice dans le tableau* `MEM` *de sa partie utile.*

Le mot d'en-tête d'une zone sera toujours un atome contenant à la fois le nombre de mots utiles de la zone et quelques bits d'information supplémentaires pouvant être utilisés si besoin par le gestionnaire mémoire. Connaissant l'adresse d'une zone, il est donc aisé de connaître sa taille utile donc l'adresse de la zone suivante.

Sur une architecture 64 bits, il est raisonnable de penser à l'heure actuelle que les zones utiles sont de taille inférieures à $2^{55} \approx 3,6 \times 10^{16}$ mots, ce qui laisse 8 bits pour ces informations en plus du bit indiquant qu'on a affaire à un atome et non à une adresse. En particulier, le gestionnaire mémoire utilise un bit de ce mot d'en-tête pour indiquer soit qu'il considère la zone comme étant libre, on dira alors qu'il s'agit d'une *zone libre*, soit qu'il considère que la zone est encore potentiellement utilisée par le programme, on dira alors qu'il s'agit d'une *zone occupée*.

Le gestionnaire mémoire doit satisfaire à tout ou partie des objectifs suivants :

1. Connaître à tout instant l'état d'occupation du tas. Cet objectif sera détaillé dans la section **2**.

2. Attribuer sur requête du programme une zone contigüe du tas pour y stocker une nouvelle donnée. Le programme spécifie quelle est la longueur en mots de la donnée à stocker et le gestionnaire mémoire renvoie en réponse à la requête un pointeur indiquant l'adresse de la zone allouée. Cet objectif sera détaillé dans la section **3**.

3. Libérer sur requête du programme une zone du tas précédemment allouée. Cette zone pourra être réutilisée pour satisfaire une requête d'allocation ultérieure. Lors de la libération d'une zone, le programme fournit comme seule information l'adresse de la zone allouée. Cet objectif sera détaillé dans la section **4**.

4. Rechercher dans le tas quelles zones ont été allouées et sont toujours accessibles par le programme. Une zone du tas est accessible s'il existe un pointeur vers cette zone stocké dans une variable locale ou globale du programme, ou stocké dans une zone du tas elle-même accessible. Les zones inaccessibles peuvent être libérées automatiquement, c'est-à-dire sans requête du programme, et réutilisées pour satisfaire une requête d'allocation ultérieure. Cet objectif sera détaillé dans la section **5**.

5. Déplacer les données dans le tas de façon à rendre contigüe la partie du tas non utilisée. Cette opération peut être nécessaire pour pouvoir satisfaire une requête d'allocation de taille supérieure à la plus grande taille des zones libres, mais inférieure à la somme de ces tailles. Cet objectif sera détaillé dans la section **6**.

Les objectifs (1) et (2) sont satisfaits par tous les gestionnaires de mémoire. L'objectif (3) est en général réservé aux langages de bas niveau pour lesquels seul le programmeur est réputé savoir à quel moment une zone allouée cesse d'être utile au programme. Les objectifs (4) et (5) sont en général réservés aux langages de haut niveau pour lesquels le programmeur est déchargé de cette responsabilité. Remarquons que pour atteindre les objectifs (4) et (5), le gestionnaire mémoire doit avoir

un moyen pour distinguer, dans le tas, les données scalaires et les pointeurs. Nous supposerons donc qu'il dispose d'une telle méthode, par exemple à l'aide d'un bit utilisé à cette fin, ce qui laisse alors $A - 1 = 63$ bits pour coder atomes et pointeurs.

Naturellement, pour fonctionner, le gestionnaire mémoire a lui aussi besoin de mémoire. Il peut utiliser la mémoire statique pour stocker les pointeurs β, γ, δ et ε. Le pointeur α est stocké implicitement ou explicitement dans le code du programme, ou dans un registre du processeur qui exécute le programme. Les autres données non statiques utiles au gestionnaire de mémoire sont stockées dans les zones libres du tas, et éventuellement dans la partie non utilisée de la pile d'exécution.

3 Connaître l'état d'occupation du tas

Une méthode simple de représentation de l'état d'occupation du tas consiste à utiliser un tableau de booléens indiquant pour chaque mot du tas s'il est libre ou non. Ce tableau est appelé *carte d'occupation* et il peut être stocké de manière compacte dans une partie du tas spécialement réservée à cet usage. Par exemple dans le cas d'un ordinateur 64 bits, chaque mot de la carte d'occupation code le statut libre/non libre de 64 mots du tas, et la carte d'occupation nécessite pour son stockage $\frac{1}{64} \approx 1,6\%$ de la taille totale du tas.

Une autre méthode consiste à lier entre elles les zones libres, à la manière d'une liste chainée. Pour cela, il est nécessaire que chaque zone libre ait, outre son mot d'en-tête, une taille utile d'au moins un mot.

Si l'on suppose que chaque zone libre a une taille utile d'au moins deux mots, alors on peut aussi représenter l'ensemble des zones libres par un arbre binaire de recherche classé par taille utile.

4 Allocation d'une zone mémoire

Il s'agit de trouver une zone libre d'au moins n mots utiles où n est la taille demandée, transmise dans la requête d'allocation, de marquer cette zone comme occupée et de retourner au programme l'adresse de la zone allouée. Pour ce faire, le gestionnaire mémoire parcourt la structure de données représentant l'état d'occupation du tas (liste chaînée, arbre binaire ou carte d'occupation) jusqu'à trouver une zone libre de taille convenable. Dans le cas général plusieurs zones libres peuvent convenir et le gestionnaire mémoire dispose d'une certaine latitude pour décider quelle zone il attribuera. Il peut allouer la première zone convenable rencontrée lors de la recherche *(allocation au premier choix)*, ou l'une de celles dont la taille utile s'approche au mieux de n *(allocation au meilleur choix)*.

Soit t la taille utile de la zone libre sélectionnée. Si $t > n$ alors les $t - n$ mots excédentaires constituent une zone libre résiduelle de taille utile $t - n - 1$ qui vient remplacer la zone de taille utile t initiale dans la structure de données représentant les zones libres. Dans le cas où les zones libres sont soumises à une contrainte de taille minimale et si t est trop proche de n cette zone résiduelle peut être inutilisable et

doit être considérée comme faisant partie de la zone allouée, bien que le programme n'en ait pas l'usage.

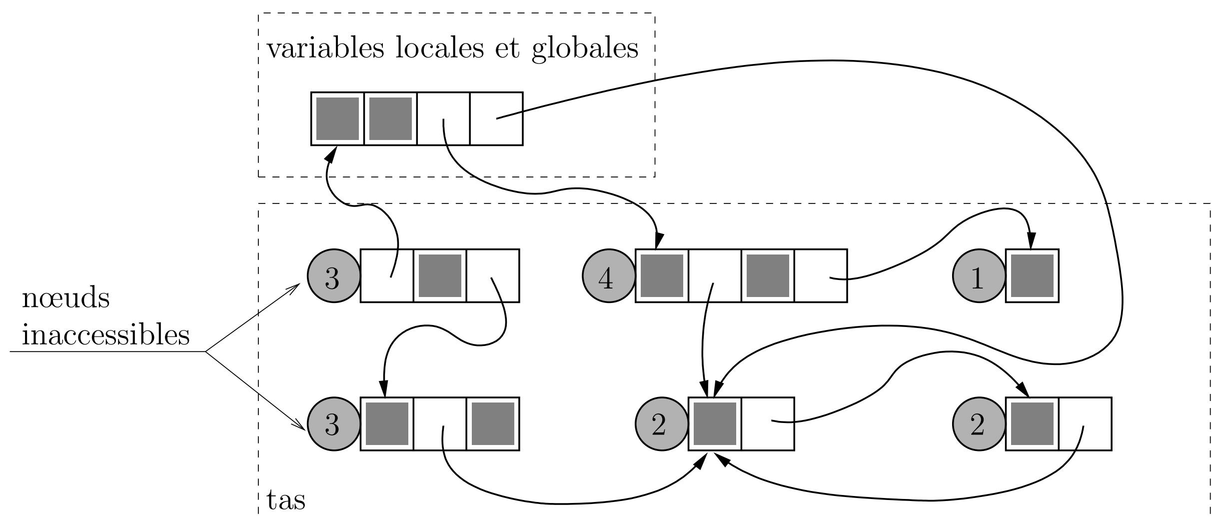
Que faire si l'on ne trouve pas de zone libre suffisamment grande ? Dans le cas d'un gestionnaire mémoire implantant uniquement les objectifs (1), (2) et (3) il n'y a plus suffisamment de mémoire contiguë disponible et la requête d'allocation échoue. Le gestionnaire mémoire signale cet échec au programme et il appartient à ce dernier de prendre des mesures appropriées : s'interrompre, réorganiser ses données pour libérer de la mémoire, ou basculer vers une exécution alternative.

Lorsque le gestionnaire mémoire implante l'objectif (4) (recherche des zones allouées encore accessibles), il lance cette recherche, libère les zones allouées qui se sont révélées inaccessibles, puis relance la requête d'allocation. Un nouvel échec à ce stade devient définitif : la requête d'allocation est non satisfiable. De même, lorsque le gestionnaire mémoire implante l'objectif (5), il réorganise d'abord le tas puis relance la requête d'allocation.

5 Libération d'une zone mémoire

La libération d'une zone mémoire pose comparativement moins de difficultés : elle ne peut pas échouer et le gestionnaire mémoire doit juste mettre à jour ses données représentant l'état d'occupation du tas. Dans le cas de la représentation par une carte d'occupation, cette mise à jour est triviale. Dans les autres cas, il faut déterminer si la zone libérée est adjacente à une ou deux zones déjà libres et le cas échéant concaténer ces zones libres adjacentes en une seule.

6 Déterminer les zones accessibles



La figure ci-dessus représente l'état de la mémoire allouée à un programme en cours d'exécution. Les cases grisées contiennent des atomes, les cases blanches d'où partent des arcs contiennent des pointeurs (on suppose que les arcs aboutissent toujours sur le premier mot d'une zone utile), et les cases rondes contiennent les en-têtes placées par le gestionnaire mémoire (le numéro représenté est la taille de la zone utile

concernée). Ainsi, la mémoire allouée au programme constitue un graphe orienté dont les nœuds sont les zones mémoire et les arcs sont les pointeurs stockés dans ces zones. Ce graphe peut comporter des cycles. Les pointeurs stockés dans les variables globales et locales sont des *points d'entrée* dans le graphe, et le gestionnaire mémoire doit calculer la fermeture transitive dans ce graphe de l'ensemble des points d'entrée pour déterminer quelles zones sont accessibles. Les nœuds alloués n'appartenant pas à cette fermeture transitive peuvent alors être libérés.

Pour déterminer quels nœuds sont accessibles, le gestionnaire de mémoire effectue un parcours du graphe en partant des points d'entrée. Chaque nœud visité est "colorié" pour ne pas être visité à nouveau si plusieurs arcs aboutissent à ce nœud. Si les nœuds sont tous initialement de couleur blanche et que lors d'une visite on colorie le nœud visité en noir, l'ensemble des nœuds accessibles est l'ensemble des nœuds noirs à la fin du parcours et tous les nœuds restés blancs, s'il y en a, sont inaccessibles et peuvent être libérés (et on colorie alors de nouveau en blanc tous les nœuds avant la prochaine recherche des nœuds accessibles).

Le gestionnaire mémoire a donc deux besoins en mémoire différents pour déterminer les nœuds accessibles : il lui faut de la mémoire pour placer les couleurs des nœuds et il lui faut de la mémoire pour stocker la liste des nœuds restant à visiter. On peut généralement utiliser les bits d'information du mot d'en-tête d'un nœud pour y placer sa couleur.

Les algorithmes classiques de parcours de graphes (en profondeur ou en largeur) utilisent une liste de nœuds à explorer sous l'une des deux formes suivantes :

- ou bien il s'agit d'un algorithme itératif et la liste des nœuds à explorer est stockée explicitement (sous forme d'une pile ou d'une file) ;
- ou bien il s'agit d'un algorithme récursif et cette liste est en fait disséminée dans la pile d'exécution du programme, dans les variables locales des différentes instances de la fonction en cours d'exécution.

Malheureusement, ces algorithmes ne sont pas directement utilisables ici car cette liste peut être particulièrement longue, le graphe pouvant comporter un grand nombre de nœuds et pouvant avoir une structure arbitraire. Les algorithmes récursifs peuvent donc ici nécessiter une profondeur d'appels récursifs trop importante pour la pile d'exécution (de taille généralement petite comparée au tas) et les algorithmes itératifs nécessiterait des allocations mémoires dans le tas. C'est ce que l'on veut éviter car c'est en général lorsqu'il n'y a quasiment plus de zones libres qu'on essaie de déterminer quelles sont les zones accessibles.

Il existe plusieurs types de solutions à ce problème :

- La première est de colorier initialement en noir les nœuds accessibles depuis la pile ou la mémoire statique, puis de parcourir séquentiellement l'ensemble des zones mémoires du tas en coloriant à chaque fois en noir les nœuds pointés par les nœuds noirs. On parcourt de nouveau l'ensemble du tas tant qu'au moins un nouveau nœud à été colorié.
- Une deuxième est d'utiliser une nouvelle couleur, le rouge. On marque en rouge les nœuds accessibles depuis la pile ou la mémoire statique. On effectue un parcours séquentiel de l'ensemble des zones du tas, et pour chaque zone

rouge, on effectue un parcours en profondeur en marquant les nœuds visités en noir. Au delà d'une certaine profondeur, on cesse d'explorer les nœuds pointés par le nœud en cours d'exploration et on marque en rouge les nœuds dont les fils n'ont pu être explorés. On recommence à parcourir l'ensemble des zones du tas jusqu'à avoir effectué un parcours complet sans avoir trouvé la moindre zone rouge.

- La troisième est d'effectuer un parcours en profondeur en utilisant une technique appelée inversion de pointeurs, beaucoup plus délicate à mettre en œuvre.

7 Réorganisation du tas

Il s'agit ici de déplacer les nœuds accessibles du graphe de sorte qu'il ne reste plus qu'une seule zone libre, contigüe. On peut envisager au moins deux stratégies :

- Faire glisser les nœuds accessibles vers l'une des extrémités du tas et mettre à jour les pointeurs vers les zones déplacées.
- Dès le départ, on convient de ne pas utiliser plus de 50% de la mémoire : le tas est divisé en deux moitiés égales et on n'alloue que dans l'une. Lors d'une réorganisation, on recopie les nœuds accessibles, de manière contiguë, vers la moitié non utilisée, puis on échange les rôles des deux moitiés. Là aussi il convient de mettre à jour les pointeurs vers les nœuds qui ont été recopiés. Cette stratégie a un coût : seule la moitié de la mémoire disponible est effectivement utilisable à un instant donné et des programmes peuvent être de ce fait empêchés de s'exécuter par manque de mémoire. En revanche, elle présente l'avantage de simplifier les opérations d'allocation et de libération. On alloue séquentiellement dans la moitié utile du tas, à la manière de l'allocation sur une pile, et on ignore les requêtes de libération (la mémoire qui aurait dû être libérée le sera automatiquement après recopie des zones accessibles dans l'autre moitié du tas).

8 Pistes de réflexion pour l'exposé

1. Écrire deux fonctions prenant en entrée une carte d'occupation du tas et un nombre n et retournant l'adresse d'une zone libre de taille utile supérieure ou égale à n s'il en existe une, selon les stratégies d'allocation au premier choix et d'allocation au meilleur choix. En cas d'échec, l'adresse retournée sera une adresse négative arbitraire. On supposera que la carte d'occupation est *non compactée*, c'est-à-dire que chaque élément de cette carte code la liberté/non liberté d'un seul mot du tas. On représentera cette carte comme un tableau de booléens (d'entiers en C) même si cette représentation n'est pas aussi compacte qu'une suite de bits.
2. Choisir une ou plusieurs méthodes de représentation des zones libres du tas (carte d'allocation, liste chaînées, arbres binaires) et détailler les algorithmes

d'allocation et de libération associés. On étudiera la complexité de ces algorithmes en fonction de la taille du tas et du nombre de zones libres. Dans le cas d'une représentation par arbres binaires, on pourra se poser la question du type d'arbre le plus adapté. On veillera en particulier à ce que ces algorithmes puissent tourner dans les contraintes mémoires qui sont celles du problème.

3. Étudier le problème de la gestion des zones perdues en raison de leur trop petite taille : y a-t-il moyen d'éviter ce problème ? De les récupérer lors d'une réorganisation ?

4. Détailler un algorithme de recherche des zones accessibles. On veillera en particulier à ce que cet algorithme puisse effectivement s'exécuter sans nécessiter d'autre mémoire que celle disponible dans le tas et la pile d'exécution.

5. On pourra commencer par lister les avantages et inconvénients de la deuxième stratégie de réorganisation du tas proposée puis on détaillera un algorithme. On étudiera en particulier le problème dû à la mise à jour des pointeurs, sachant que la seule mémoire disponible est celle disponible dans le tas et la pile d'exécution.

6. Le texte suppose implicitement que le gestionnaire mémoire et le programme fonctionnent "à tour de rôle". Étudier comment on peut lever cette hypothèse et obtenir un fonctionnement parallèle. En discuter les avantages et les inconvénients.

7. D'un point de vue éthique, la discussion peut s'orienter vers au moins deux directions :

   - la sécurité : que se passe-t-il si l'allocation ou la libération mémoire sont incorrectement faites ; qui est responsable des problèmes induits ?
   - les conséquences environnementales avec une réflexion sur l'efficacité de ces algorithmes, cruciale dans certaines situations.