International Journal of Application or Innovation in Engineering & Management (IJAIEM)
Web Site: www.ijaiem.org Email: editor@ijaiem.org
Volume 6, Issue 7, July 2017 **ISSN 2319 - 4847**

Nouvel Algorithme pour gérer la mémoire dans les systèmes embarqués : applications java

Laila FAL[1], Laila MOUSSAID[2], Hicham MEDROMI[3], Aroua Amari[4]

Equipe Architecture des systémes ENSEM Casablanca, Maroc

RESUME

La gestion manuelle de la mémoire demande au programmeur de déclarer explicitement toutes les allocations et les dés allocations, c'est pour cela, des nombreux algorithmes ont été mis en place pour améliorer la gestion dynamique de la mémoire dans les systèmes embarqués. En java, la gestion de mémoire est effectuée par des ramasse-miettes, ils interrompent le programme de temps à autre pour examiner la mémoire. Ces temps de pause imposés à l'exécution à des moments impossibles à prédire, et pour des durées inconnues rendent le comportement du programme imprévisible, en plus de la fragmentation de la mémoire. Suites aux études effectuées, chaque algorithme de gestion convient à des applications précisent, il n'existe pas un algorithme généralisé idéal qui peut être appliqué sur tous les types d'applications, c'est pour cela que nous avons proposé dans cet article un algorithme hybride qui combine les deux approche de gestion en régions et par générations afin de remédier aux problèmes rencontrés et améliorer la gestion de la mémoire dans les systèmes embarqués.

Mots clés : Systèmes embarqués, ramasse-miettes, gestion de la mémoire en régions, gestions de la mémoire par génération.

1. INTRODUCTION

Les besoins en termes de mémoire ont devenu de plus en plus importants dans les systèmes embarqués, ce qui a amené les industriels à adopter des techniques de développement logiciel spécifiques au monde des systèmes embarqués.
Le langage Java offre un avantage considérable en termes de facilité d'utilisation : la gestion transparente de la mémoire dynamique, basée sur l'emploi de ramasse-miettes. Or, les algorithmes couramment utilisés pour implanter ces mécanismes introduisent des temps de pause dans l'exécution des programmes.
Il existe des multitudes d'algorithmes, chacun convient plus à un type précis d'applications.
Dans cet article, je commence par la présentation des différentes notions utilisées tels que ramasse-miettes et systèmes embarqués, puis je passe à l'exposition des algorithmes classiques existants pour détailler leurs avantages et inconvénients, ensuite je présente la nouvelle approche proposée pour résoudre les problèmes de fragmentation et temps de pauses produits par les ramasse-miettes et je termine par une conclusion suivie des perspectives.

2. CONTEXTE ET ETAT DE L'ART

2.1 Systèmes embarqués

Un système embarqué [7] peut être défini comme un système électronique et informatique autonome, qui est dédié à une tâche bien précise, possédant des ressources d'ordre spatial (taille limitée) et énergétique (consommation restreinte) limitées. Le terme de « Système Embarqué » désigne aussi bien le matériel que le logiciel utilisé.
Les systèmes embarqués exécutent des tâches prédéfinies et ont un cahier des charges contraignant à remplir, qui peut être d'ordre :

- D'espace compté, avec un espace mémoire limité de l'ordre de quelques Mo maximum. Ils font de plus très souvent appel à l'informatique et aussi aux systèmes « temps réel ».
- De consommation énergétique le plus faible possible, due à l'utilisation de sources autonomes, batteries, panneaux solaires
- Temporelle, dont le temps d'exécution de tâches est déterminé
- Sûreté de fonctionnement qui demande aux systèmes dits critiques de fournir des résultats exacts et pertinents [8]
- Sécurité indispensable pour assurer la confidentialité des données utilisées, notamment pour les systèmes employés au service de la santé [8].

3.1 Ramasse-miettes

Le ramasse-miettes est une fonctionnalité de la JVM qui a pour rôle de gérer la mémoire notamment en libérant celle des objets qui ne sont plus utilisés [3].
La règle principale pour déterminer qu'un objet n'est plus utilisé est de vérifier qu'il n'existe plus aucun autre objet qui lui fait référence. Ainsi un objet est considéré comme libérable par le ramasse-miettes lorsqu'il n'existe plus aucune référence dans la JVM pointant vers cet objet.
Lorsque le ramasse-miettes va libérer la mémoire d'un objet, il a l'obligation d'exécuter un éventuel finalizer définit dans la classe de l'objet. Attention, l'exécution complète de ce finalizer n'est pas garantie : si une exception survient durant son exécution, les traitements sont interrompus et la mémoire de l'objet est libérée sans que le finalizer soit entièrement exécuté.

La mise en œuvre d'un ramasse-miettes possède plusieurs avantages :

- Elle améliore la productivité du développeur qui est déchargé de la libération explicite de la mémoire.
- Elle participe activement à la bonne intégrité de la machine virtuelle: une instruction ne peut jamais utiliser un objet qui n'existe plus en mémoire.

Mais elle possède aussi plusieurs inconvénients :

- Le ramasse-miettes consomme des ressources en terme de CPU et de mémoire.
- Il peut être à l'origine de la dégradation plus ou moins importante des performances de la machine virtuelle.
- Le mode de fonctionnement du ramasse miettes n'interdit pas les fuites de mémoires si le développeur ne prend pas certaines précautions. Généralement issues d'erreurs de programmation subtiles, ces fuites sont assez difficiles à corriger.

3.2 Gestion de la mémoire

3.2.1 Les algorithmes de ramasse-miettes classique

Il existe plusieurs classes d'algorithmes principales permettant la gestion automatique de la mémoire dynamique [6]. Le ramasse-miettes par comptage de références, puis celui par marquage-balayage et, enfin, celui par recopie seront décrits.

Le ramasse-miettes par comptage de références :

La technique dite du «comptage de références» [1] consiste à associer à chaque objet un compteur, représentant le nombre de références existantes sur cet objet. Le compteur d'un objet est incrémenté lors de l'apparition d'une nouvelle référence sur cet objet ; il est décrémenté lorsqu'une de ces références disparaît : si le compteur devient nul, l'objet n'est plus référencé et la mémoire qu'il occupe peut alors être récupérée.

Cet algorithme présente l'avantage d'être très simple à mettre en œuvre : il suffit pour cela d'ajouter un champ à tous les objets, et de détecter l'apparition, la modification, ou la suppression d'une référence (une barrière en écriture est une portion de code s'exécutant à chaque écriture de référence ; le surcoût occasionné par ce genre de mécanisme peut généralement être très réduit [5]).

De plus, il peut permettre une allocation en temps prédictible, puisque la mémoire est récupérée dès la mort d'un objet. En outre, la libération se faisant récursivement, elle a un coût d'exécution au pire cas proportionnel au nombre d'objets du tas (libération de tout son contenu). Une adaptation présentée plus loin propose une réduction de ce coût.

En revanche, il présente une limitation importante : il est inopérant sur les données cycliques. La figure 1 illustre cet inconvénient majeur, qui impose alors d'associer ce type de ramasse-miettes avec un autre mécanisme, destiné à libérer la mémoire de ce genre de structures. En outre, cette méthode de récupération de la mémoire n'empêche pas la fragmentation du tas : une allocation peut échouer alors que la quantité de mémoire libre est suffisante, mais qu'elle est trop fragmentée.

Enfin, la place occupée en mémoire par le compteur peut avoir un impact sur l'occupation de la mémoire, ressource souvent rare dans les systèmes qui nous intéressent [1].

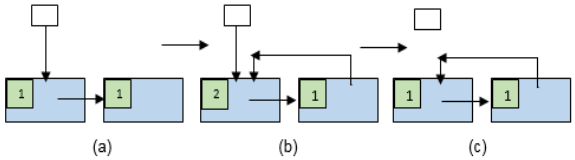

Figure 1 Illustration du problème de la gestion des cycles par un ramasse-miettes de type comptage de références. Deux éléments d'une structure cyclique ont initialement leur compteur à 1 (ils ne sont référencés qu'une seule fois chacun) (a). Le cycle est alors construit : le premier élément est référencé deux fois (b). Enfin, l'unique pointeur sur la structure complète disparaît, mais les compteurs ne sont pas nuls (c) : la place occupée par les deux éléments est définitivement perdue [1].

Le ramasse-miettes par marquage-balayage :
Une autre technique consiste à exécuter un algorithme de libération de mémoire lorsqu'une tentative d'allocation échoue (i.e. il n'y a plus de zone de mémoire inoccupée de taille suffisante pour contenir le nouvel objet) [1].

Un ramasse-miettes par marquage-balayage [1] consiste à différencier l'ensemble des objets accessibles du reste du tas. Il s'effectue à partir d'un état initial dans lequel aucun élément n'est marqué.

Ces objets sont alors parcourus et marqués récursivement à partir des références dites racines (pile d'exécution, etc.). Enfin, ceux qui n'ont pas été visités sont considérés comme inaccessibles, et la mémoire qu'ils occupent est récupérée.
Contrairement à l'algorithme par comptage de références [1], celui-ci ne nécessite pas de barrière en écriture. Il permet de plus la récupération des structures cycliques. Le surcoût en taille mémoire nécessaire peut également être réduit, puisqu'un seul bit suffit à stocker l'information de marquage d'un objet.

En revanche, le temps au pire cas de l'allocation devient beaucoup plus important, dans la mesure où il nécessite au moins deux parcours de l'ensemble des objets. Ce genre de mécanisme peut alors occasionner des pauses de durées considérables pendant l'exécution d'un programme.
Enfin, il n'empêche pas la fragmentation de la mémoire.

Le ramasse- ramasse-miettes par recopie :
L'algorithme de ramasse-miettes par recopie [1] est basé sur un partitionnement du tas en deux zones de tailles égales. Les allocations se font en incrémentant un pointeur dans une première zone (donc en temps constant), jusqu'à rencontrer la fin de celle-ci. L'algorithme se déclenche alors, parcourant les objets de la première zone et les recopiant dans la seconde. Enfin, le rôle des deux zones est interverti.

Un inconvénient à cette recopie est la nécessité d'utiliser des barrières en lecture (i.e. mécanisme d'indirection exécuté lors de chaque lecture d'une référence), qui ont pour particularité d'être assez coûteuses en temps d'exécution. En effet, le déplacement des objets par le ramasse-miettes pendant l'exécution du programme implique au moins un degré d'indirection lors de la lecture d'une référence. Hormis cet aspect, cet algorithme présente à peu près les mêmes propriétés qu'un ramasse-miettes par marquage-balayage. Il élimine toutefois la fragmentation de la mémoire, en contrepartie d'un doublement de l'espace mémoire nécessaire à l'exécution d'un même programme.

Des versions améliorées de ces différents mécanismes existent déjà, mais les avancées sur un point se font souvent au détriment d'un autre. Par exemple, Hudson et Moss [4] ont proposé un algorithme par recopie réduisant la taille de la mémoire supplémentaire nécessaire, mais cet algorithme devient très peu efficace s'il doit traiter de grandes structures cycliques.

Table 1: Comparatif des algorithmes classiques

Méthode de gestion	Avantages	Inconvénients
Comptage de références	simple à mettre en œuvre -Allocation en temps prédictible	Nécessite une barrière en écriture Inopérant sur les données cycliques

		N'empêche pas la fragmentation du tas
Marquage-balayage	Ne nécessite une barrière en écriture récupération des structures cycliques Surcoût du stockage de l'information de marquage d'un objet réduit	Pauses de durées considérables pendant l'exécution du programme N'empêche pas la fragmentation du tas
Par recopie	Empêche la fragmentation du tas Couts de déplacement des objets	Nécessite une barrière en écriture Doublement d'espace mémoire pour l'exécution du programme

3.2.1 La gestion de la mémoire par région

Les objets qui ont le même cycle de vie sont alloués dans une même région. L'espace mémoire occupé par une région peut être libéré lorsque tous les objets qu'elle contient sont morts [1].

Les avantages de cette technique de gestion de la mémoire sont divers. Elle permet d'une part de réduire l'espace nécessaire pour exécuter une application. Effectivement, ce dispositif permet en principe de récupérer la mémoire occupée par les objets morts relativement tôt ; un groupe d'objet (liste, arbre, etc.) dont tous les éléments sont placés dans la même région, peut être libéré en une seule fois lorsque la dernière référence sur cette structure disparaît. De plus, les primitives de gestion de la mémoire peuvent être implémentées de manière à s'exécuter en des temps bornés [1].

Il existe différents types de région, à taille fixe ou variable. Une taille de région fixe implique de pouvoir borner le nombre et la taille des objets qui y seront alloués, alors qu'une taille variable apporte plus de liberté, au prix d'une gestion plus coûteuse.

Par contre, ce procédé implique de pouvoir déterminer les objets qui ont un cycle de vie similaire.
On peut confier cette tâche au programmeur ou tenter de l'automatiser grâce à une analyse statique [1].

La gestion de la mémoire par génération [2]

Un ramasse-miettes générationnel distingue plusieurs zones différentes dans le tas : la crèche (ou nursery), où sont alloues les nouveaux objets, et une ancienne génération (old generation, ou mature space), ou se trouveront les objets réputés ≪vieux≫.

Les allocations se font dans la crèche, par un simple décalage de curseur. Lorsque l'espace est épuise, le ramasse-miettes entreprend une collecte mineure : il parcourt la crèche en commençant par les racines, et déplace les objets vivants dans l'ancienne génération. Cette opération est appelée titularisation (tenuring) : si des objets ont survécu jusqu'à la collecte mineure, ils sont titularisés dans la zone principale.

La collecte des objets morts de la crèche requiert bien sur un calcul d'accessibilité dans le graphe du tas tout entier, et donc le ramasse-miettes doit tenir compte des pointeurs entre les générations. Pour ne pas avoir à parcourir l'ensemble du tas à chaque collecte mineure, les pointeurs provenant de l'ancienne génération font l'objet d'un traitement spécial : ils sont considérés directement comme des racines lors du parcours de la crèche. Pour détecter ces pointeurs, un ramasse-miettes générationnel a besoin d'une barrière en écriture : à chaque fois que le programme crée un pointeur

d'un ≪vieil≫ objet vers un ≪jeune≫ objet, l'environnement d'exécution copie ce pointeur dans une zone dédiée que le ramasse-miettes ajoutera à son ensemble de racines lors de la phase de marquage.

Par exemple, dans la figure 2, l'espace libre de crèche est épuisé, et le ramasse-miettes va entamer une collecte mineure. Les deux seules racines du tas sont les variables r1 et r2, mais la barrière en écriture a intercepté l'écriture dans o5 du pointeur vers o7, et a copié le pointeur dans les ≪racines de la crèche≫. Par contre, le pointeur de o8 vers o6 n'est pas intéressant : o6 est déjà dans l'ancienne génération, et donc n'est pas affecté par les collectes mineures.

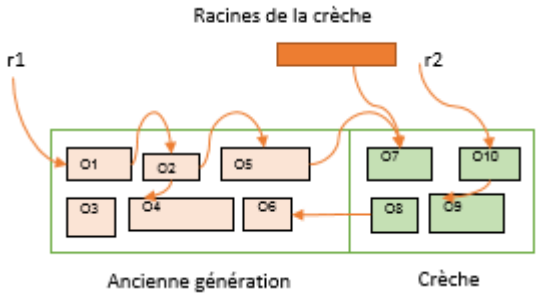

Figure 2 Ramasse-miettes générationnel. La crèche est entièrement occupée, et va subir une collecte mineure. Les objets survivants seront déplacés dans l'ancienne génération [2].

La crèche est collectée en utilisant une technique de copie : au fur et à mesure du parcours, les objets vivants (o7, o9, o10) sont déplacés dans l'ancienne génération. Le tas obtenu après la collecte est représenté sur la figure 3 : l'espace de la crèche est maintenant considéré comme libre (o8 a été libère implicitement lorsque le pointeur sommet de la crèche a été réinitialisé) et le, programme utilisateur peut être relance. On remarquera que o3 et o6 n'ont pas été collectes : ce sont des objets morts de l'ancienne génération, et il faudra attendre la prochaine collecte majeure, lors de laquelle l'ensemble du tas sera parcouru, pour s'apercevoir qu'ils sont inaccessibles.

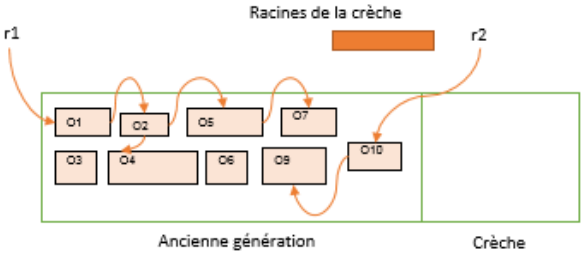

Figure 3 Ramasse-miettes générationnel après une collecte mineure. La crèche est entièrement disponible, et tous les objets restants sont situés dans l'ancienne génération [2].

3. NOUVELLE APPROCHE PROPOSEE

La nouvelle approche combine l'algorithme générationnel avec celui de gestion par région, sauf que mon algorithme ne va pas imposer des hiérarchies entre les générations (crèche et vieux) mais les objets seront alloués dans des régions.

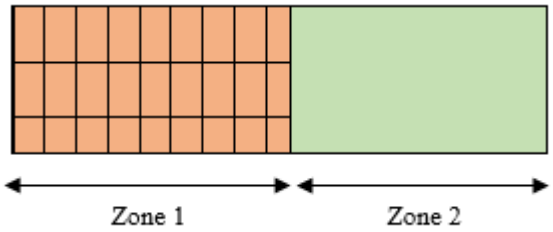

Figure 4 Découpage du tas en deux zones

En fait, il s'agit de découper le tas en deux zones, puis les allocations se feront par région, les objets ayants la même durée de vie seront alloués dans la même région.
Les allocations se feront dans la première zone, dès que nous avons une demande d'allocation non satisfaite, c'est à dire mémoire insuffisante, il faut lancer une collection sur la première zone.
Le but de cette collection est de déterminer les objets vivants et les déplacer par régions vers la deuxième zone, ensuite l'espace mémoire occupé par les régions contenant des objets morts sera récupéré.
Si la deuxième zone est remplie, déclencher une collection (un ramasse miette simple) sur celle-ci pour récupérer de l'espace mémoire.

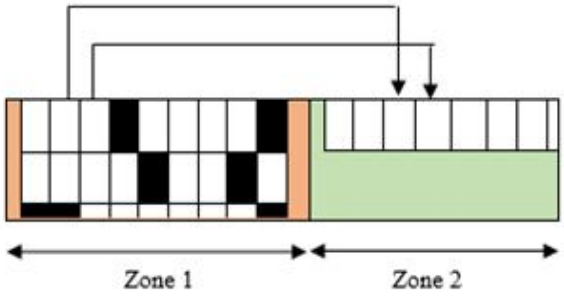

Figure 5 Déplacement des objets vivants vers la zone 2

Cette nouvelle approche permettra de:

- Collecter rapidement au moment opportun chaque zone
- Réduire le nombre de collecte à effectuer
- Gain du temps d'exécution, car en découpant le tas en deux zones, la collecte sur une zone sera rapide
- Libérer un tas de mémoire en une seule collection
- Classer les objets par durée de vie
- Réduire les couts liés à la copie et déplacement des objets puisque plusieurs objets seront traités au même temps.

4. CONCLUSION ET PERSPECTIVES

L'étude des différentes solutions existantes au problème de la gestion de la mémoire dynamique a permis de mettre en avant quelques problèmes et les diverses solutions qui ont été proposées pour les résoudre. À partir de ces observations, j'ai proposé une autre approche hybride qui combine deux algorithmes de gestion en régions et par génération afin de réduire le temps d'exécutions et de pauses.
Enfin, pour poursuivre, une perspective serait d'implémenter cet algorithme dans une variante java, désactiver le ramasse-miettes et le remplacer par notre nouvel algorithme puis le lancer sur plusieurs applications et comparer/analyser les résultats obtenus.

References

[1]. Nicolas Berthier : Gestion hybride de la mémoire dynamique dans les systèmes Java temps-réel
[2]. Guillaume Salagnac : Synthèse de gestionnaires mémoire pour applicationsJava temps-réel embarquées
[3]. Jean-Michel DOUDOUX:Site https://www.jmdoudoux.fr/java/dej/chap-gestion_memoire.htm publié en 2016
[4]. Richard L. Hudson and J. Eliot B. Moss. Incremental collection of mature objects. In IWMM '92 : Proceedings of the International Workshop on Memory Management, pages 388–403, London, UK, 1992. Springer-Verlag
[5]. Antony L. Hosking, J. Eliot B. Moss, and Darko Stefanovic. A comparative performance
[6]. evaluation of write barrier implementation. In OOPSLA '92 : conference proceedings on
[7]. Object-oriented programming systems, languages, and applications, pages 92–109, New York,
[8]. NY, USA, 1992. ACM Press. (cité page 7)
[9]. Richard Jones and Rafael Lins. Garbage collection : algorithms for automatic dynamic memory
[10]. management. John Wiley & Sons, Inc., New York, NY, USA, 1996. (cité page 7)
[11]. Corinne Alonso, Bruno Estibals : « Conception et Commande de Systèmes Electriques Embarqués »
[12]. http://www.futura-sciences.com/tech/definitions/technologie-systeme-embarque-15282/

Authore

Laila FAL chercheur, Equipe Architectures des Systémes, Ecole Nationale supérieur d'électricité et de mécanique Route d'El Jadida. BP 8118 Oasis Casablanca, Maroc

Laila MOUSSAID Professeur chercheur, Equipe Architectures des Systémes, Ecole Nationale supérieur d'électricité et de mécanique Route d'El Jadida. BP 8118 Oasis Casablanca, Maroc

Hicham MEDROMI Professeur chercheur, Equipe Architectures des Systèmes, Ecole Nationale supérieur d'éléctricité et de mécanique Route d'El Jadida. BP 8118 Oasis Casablanca, Maroc

Aroua AMARI chercheur, Equipe Architectures des Systémes, Ecole Nationale supérieur d'éléctricité et de mécanique Route d'El Jadida. BP 8118 Oasis Casablanca, Maroc