社団法人 情報処理学会　研究報告
IPSJ SIG Technical Report

2004－MBL－31　(3)
2004－ＩＴＳ－19　(3)
2004／11／11

携帯端末における複数 Java アプリケーション起動方法の検討

前田　慎司*、岡田　英明*、清原　良三*

携帯端末の生産性向上を目的として、アプリケーションを Java で実装した場合、複数の Java アプリケーションを同時に実行する機能が必要となる。携帯端末はメモリ容量に制限があるため、特にメモリ管理が重要となる。

本論文では単一の JavaVM 上で動作する複数のアプリケーションへのメモリ割り当てとガーベージコレクション(以下、GC)の方式としてメモリブロック化方式を提案する。メモリブロック化方式はメモリを複数のブロックに分割し、ブロック単位でアプリケーションにメモリを割り当てることにより、アプリケーションへの柔軟なメモリ割り当てを実現しながらアプリケーション単位の GC を可能とする。本方式により 1 回の GC 実行時間を短縮し、GC が他のアプリケーションの実行に与える影響を低減できる。

Execution of Multiple Java Application for Mobile Terminal

Shinji MAEDA*, Hideaki OKADA*, Ryozo KIYOHARA*

Java programming language has high productivity and is attractive for developing mobile applications. Suppose most of mobile applications are written in Java, we need the mechanism for executing multiple Java applications at the same time. Since mobile terminals do not have sufficient memory, memory management is important.

This paper presents the design of Memory-Block Allocation, a new technique of memory allocation and garbage collection for executing multiple Java applications on a single Java Virtual Machine. Memory-Block Allocation makes it possible to allocate memory flexibly and to reduce the elapsed time for one garbage collection.

1. はじめに

携帯端末の高機能化に伴い、アプリケーションソフトウエアの開発規模が急増している。現在、携帯端末のアプリケーションは大半が C/C++等で実装されている。これをコードの再利用性が高く、自動メモリ管理機能を備える Java で実装することにより、生産性を向上させることができる。

アプリケーションの大半を Java で実装した場合、複数の Java アプリケーションを同時に実行する機能が必要になる。しかし、1 つの Java アプリケーションにつき 1 つの JavaVM を起動し、それぞれ別プロセスとして実行することは、CPU 性能やメモリ容量が制約されている携帯端末では難しい。

本論文では携帯端末において、複数の Java アプリケーションを同時に実行する際の課題を検討し、その解決策として単一の JavaVM 上で動作する複数のアプリケーションへのメモリ割り当てとガーベージコレクション(以下、GC)の新方式を提案する。

2. 複数アプリケーション実行の検討

2.1. 複数アプリケーションの利用例

本節では複数アプリケーションの同時実行に関する課題抽出を目的として、携帯端末上で複数のアプリケーションを利用する例を想定する。

*三菱電機(株)　情報技術総合研究所
Mitsubishi Electric Corporation
Information Technology R&D Center

表 1: アプリケーションの特性

	ﾒﾓﾘ	CPU	応答	起動	背面
メーラ	大	中	中	短	有
ブラウザ	中	中	中	中	有
ｽｹｼﾞｭｰﾗ	中	中	中	短	有
電卓	小	小	中	中	無
電話帳	小	小	中	短	無
ゲーム	大	大	高	中	無

携帯端末ではプリインストールアプリケーションをはじめとして、アプリケーションベンダの製品やユーザ自身が作成したソフトウェア等、様々なアプリケーションが利用される。携帯端末の典型的なアプリケーションについて、その特性としてメモリと CPU の消費量の大きさ、要求される応答性能、要求される起動時間、画面が背面に移動した状態での処理の有無を表 1に示す。

これらのアプリケーションはいずれも GUI を持ち、ユーザ入力に応じた処理と画面描画を行っている。例えば、電卓や電話帳はリソース消費量が少なく、性能面の要求も少ない軽量アプリケーションである。

メーラやブラウザは標準的なリソース消費量であるが、ユーザが使用したい時にすぐに起動する必要がある。さらに、メーラは前面から背面に移動した状態でも、通信やメール振り分け等の処理を継続する必要がある。

ゲームは最大限のリソースを消費して実行することを前提としていることが多く、実行中に背面のアプリケーションや GC 処理の影響を受けると操作性を著しく損なう。

次節では、これらのアプリケーション特性を考慮し、複数アプリケーション同時実行の課題を抽出する。

2.2. 課題

複数のアプリケーションが同時に動作する典型例として、前面でゲームのようにリソース消費量が多く、高い応答性能が必要なアプリケーションが動作し、背面でメモリ消費量が多いメーラが動作していることを想定し、以下の 2 つの課題を検討する。

a) リソースの節約
b) 他アプリケーションの影響の抑制

2.2.1. リソースの節約

携帯端末は CPU 性能とメモリ容量に制約があるため、複数アプリケーションの同時実行にはリソースの節約が必要である。

GUI を持つアプリケーションの場合、前面のアプリケーションの操作性がユーザの利便性に大きく寄与する。このため、対症的な解決策としては、背面アプリケーションに与えるリソースを最小限に抑え、前面アプリケーションに多くのリソースを与える方法がある。

また、実際にリソースを節約する方法としては、複数の JavaVM 間で共通クラスを共有する方法や単一の JavaVM で複数のアプリケーションを動作させる方法がある(2.3 節参照)。

2.2.2. 他アプリケーションの影響の抑制

複数のアプリケーションを同時に実行する場合、あるアプリケーションが他のアプリケーションに影響を与えることがある。

例えば、あるアプリケーションに対して GC を実行した場合、その CPU 負荷により、他のアプリケーションの実行が妨げられる。特に高い応答性能を必要とするゲーム等を実行している際には GC の実行に配慮が必要である。

また、リソースの節約を目的として、単一の JavaVM で複数のアプリケーションを動作させた場合、あるアプリケーションの不具合により JavaVM が異常終了し、それが原因で他のアプリケーションが終了するという問題がある。

この解決策としては、JavaVM をシステムアプリケーションとユーザアプリケーションの 2 プロセス構成にする方法がある。

すなわち、システムアプリケーション側では、プリインストールアプリケーション等の信頼性が高く、かつリソース消費量の少ないアプリケーションを動作させる。

一方、ユーザアプリケーション側では、信頼性が低いアプリケーションや複雑な描画処理を必要とするリソース消費量の多いアプリケーションを動作させる。この場合、ユーザアプリケーション側では、単一の JavaVM 上でリソース消費量の多いアプリ

ケーションを同時に動作させる機構が必要となる。

2.3. 関連研究

複数のアプリケーションの同時実行に関する研究として、JavaVM 間で共通するクラスのデータを共有する Class Data Sharing(以下、CDS)[1]や単一プロセス上で複数の Java アプリケーションを同時に動作させる Multi-tasking Virtual Machine(以下、MVM)[2]がある。

本節ではこれらの関連研究を紹介する。

2.3.1. CDS

CDS は JavaVM が共通で使用するクラスのデータを共有メモリに配置し、JavaVM プロセス間で共有する技術である。

SUN 社の J2SE1.5[3]では CDS が導入されており、複数のアプリケーションがシステムクラスを共有する。その結果、共有クラスのロード処理は初回利用時だけで 2 回目以降不要となり、メモリリソースの節約と起動時間の短縮を実現している。

ただし、単一のプロセス上で複数の Java アプリケーションを実行することには未対応であるため、プロセスの実行に最低限必要なリソース等を節約することはできない。

2.3.2. MVM

MVM は CDS を発展させた技術であり、単一プロセス上で複数 Java アプリケーションの同時実行を可能とする。

JSR-121[4]では単一プロセス上で複数の Java アプリケーション(JSR-121 では Isolate と呼ぶ)を独立して動作させるための仕様、Isolate 間の通信仕様、及び Isolate のライフサイクル制御を規定している。

各 Isolate はロードされたコードや JIT コンパイル済みのネイティブコードを共有する。一方、システムプロパティ、クラスパス、スタティック変数等の共有できないデータは個別に保持する。これにより、独立したプロセス上でのアプリケーションと同様に、単一プロセス上で複数の Isolate を動作させることを実現している。

MVM のメリットとデメリットを以下に列挙する。

a) メリット

・起動時間の短縮
2 番目に起動されたアプリケーションは共通クラスのロード処理が不要になり、起動時間が短縮される。

・メモリリソースの節約
読込専用データを共有することにより、メモリの消費量を抑えることができる。

・性能向上
他のアプリケーションで動的コンパイル済みの共通クラスは再コンパイル処理が不要になり、性能が向上する。

b) デメリット

・他アプリケーションの影響
JavaVM を共通化することにより、1 つのアプリケーションの不具合が他の全てのアプリケーションに影響を与える可能性がある。

2.3.3. GC

GC は Java に実装されている自動メモリ管理機能であり、古くから各種研究が行われている[5]。携帯端末向け Java に採用されている GC 方式を以下に紹介する。

SUN 社の KVM[6]が採用しているマーク&スイープ方式[7]は以下の 3 つの工程により GC を実行する。

a) マーク工程
アプリケーションが参照している全てのデータにマークを付ける。マーク付けされなかったデータは不要とみなす。

b) スイープ工程
マーク工程でマーク付けされなかった不要データを回収し、空き領域とする。

c) コンパクション工程
必要データを再配置し、空き領域を再利用可能な連続領域とする。

SUN 社の CLDC HI[8]ではマーク&スイープ方式を改良した世代別 GC[9]を採用している。世代別 GC は「データの大半はごく短期間だけ必要とされる」という特性を利用した方式であり、データの存在期間に注目し、新しいデータを新世代領域に、古いデータを旧世代領域にそれぞれ割り当て、新世代領域に対する GC の実行頻度を高くし、全体的に GC の効率を向上させている。

携帯端末向け Java に適用されていない代表的な GC 方式として参照カウント方式[10]がある。参照カウント方式は全てのデータの被参照数をカウントし、非参照数が 0 になると不要データとして直ちに回収する。

しかし、参照カウント方式はデータの参照関係に変更が生じる度にカウンタを変更するオーバヘッドが大きく、また不要データが相互参照している場合に回収不能となるという問題がある。本論文では参照カウント方式を検討対象外とする。

3. メモリブロック化方式の提案

本章では複数のアプリケーションがメモリを共同利用する場合において、従来の GC 方式を適用する際に発生する問題を示し、その解決策としてアプリケーションへの柔軟なメモリ割り当てと効率的な GC の実行を実現するメモリブロック化方式を提案する。

3.1. 従来の GC の問題点

従来の GC 方式は単一 JavaVM 上で複数のアプリケーションを動作させることを想定していない。このため、複数のアプリケーションがメモリを共同利用する環境に適用した場合に問題が生じる。

メモリ領域内にアプリケーション 1 とアプリケーション 2 のデータが混在している状態で(図 1)、アプリケーション 1 に対してマーク&スイープ方式で GC を実行する例を以下に示す。

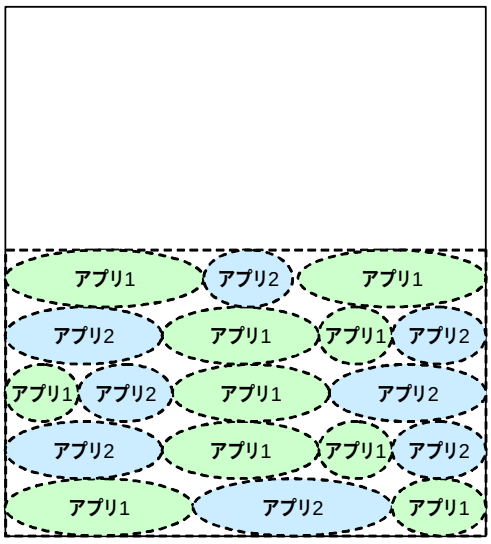

図 1: GC 実行前

まず、マーク工程でアプリケーション 1 の必要データにマークを付ける(図 2)。しかし、アプリケーション 2 の必要データにマークを付けていないため、アプリケーション 1 の不要データとアプリケーション 2 の必要データの見分けができず、アプリケーション 1 の不要データを回収できない。

従って、メモリ領域内のデータが必要か不要かを判定するには、メモリ領域を利用している全てのアプリケーションに対するマーク工程を実行しなければならない(図 3)。

結果として、不要データを回収し、連続した空き領域を作るには長時間掛かる可能性がある(図 4)。

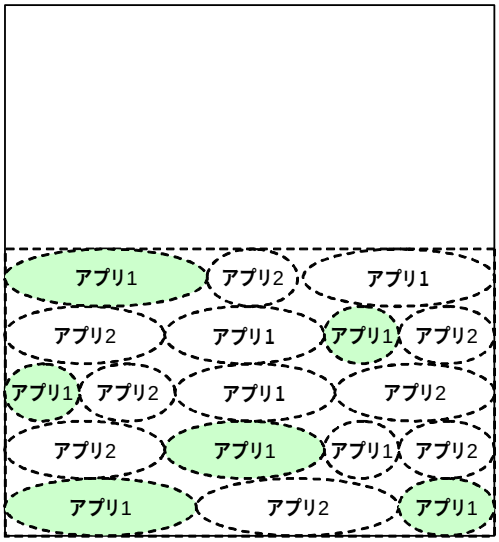

図 2: アプリ 1 のマーク実行後

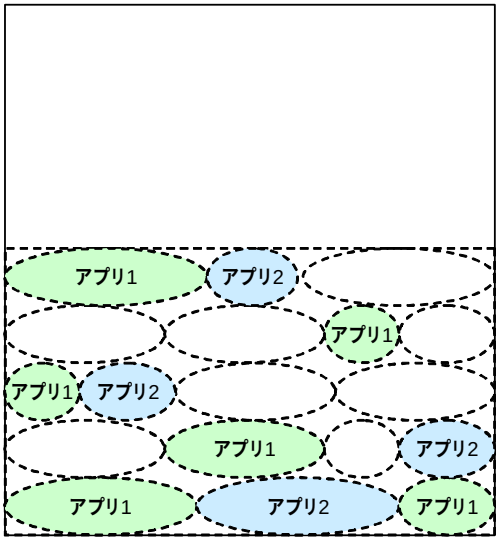

図 3: アプリ 1 とアプリ 2 のマーク実行後

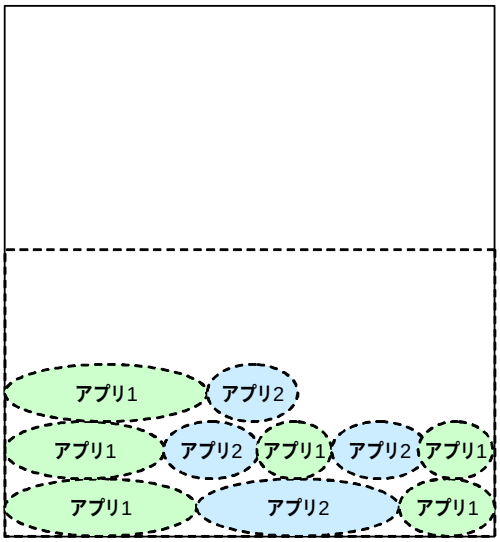

図 4: コンパクション実行後

この対策として、アプリケーションが使用する全てのデータについて、どのアプリケーションが使用しているデータかを管理し、スイープ工程で不要データを回収する際にマーク付けされていないデータを使用するアプリケーションを特定する方法がある。

しかし、全てのデータに管理領域が必要であり、データ管理処理のオーバヘッドを生じるため、有効ではない。

3.2. メモリブロック化方式の提案

メモリブロック化方式はメモリ領域を複数のブロックに分割し、JavaVM がアプリケーションに対してブロック単位でメモリを割り当てる(図 5)。

3.3. メモリ割り当て

アプリケーションに割り当てられたブロックはブロック ID、アプリケーション ID、ブロックの開始アドレス、終了アドレスからなるブロック管理テーブルによって管理される(表 2)。

アプリケーション実行中のデータ生成等により、アプリケーションに割り当て済みのブロック内の空き領域が枯渇した場合、JavaVM はアプリケーションに新たなブロックの割り当てを試みる。

ブロック割り当ての際、JavaVM はブロック管理テーブルを参照し、未使用ブロックがある場合、アプリケーションに未使用ブロックを割り当て、アプリケーションにブロック

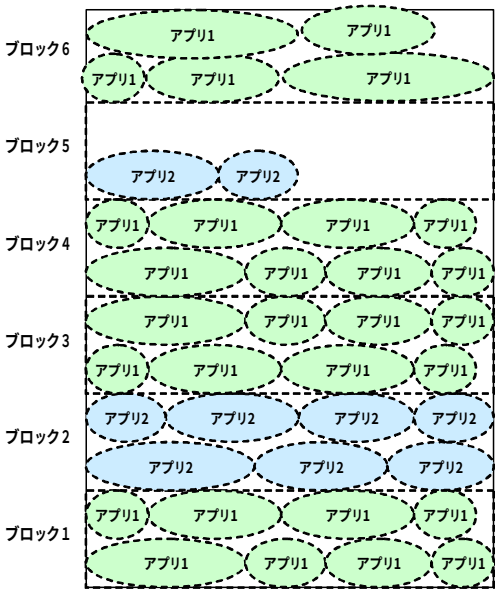

図 5: メモリのブロック化

表 2: ブロック管理テーブル

ブロックID	アプリID	開始番地	終了番地
1	1	0x0000	0x1FFF
2	2	0x2000	0x3FFF
3	1	0x4000	0x5FFF
4	1	0x6000	0x7FFF
5	2	0x8000	0x9FFF
6	1	0xA000	0xBFFF

を割り当てたことをブロック管理テーブルに登録する。未使用ブロックがない場合、GC を実行することにより未使用ブロックを生成する。

3.4. GC の実行

アプリケーション 1 とアプリケーション 2 が、割り当てられたブロックにデータを配置している例を図 5に示す。また、表 2 のブロック管理テーブルは、アプリケーション 1 にブロック 1、3、4、6 の 4 つ、アプリケーション 2 にブロック 2、5 の 2 つが割り当て済みとして登録されている例である。

ここでアプリケーション 1 がメモリを要求し、アプリケーション 1 に対して GC を実行する例を説明する。まず、アプリケーション 1 が新たなメモリを要求すると、未使用ブロックがないため、JavaVM はアプリケーション 1 に対する未使用ブロックの割り当てに失敗する。ここではアプリケーション 1 の実行中にブロック割り当てに失敗したた

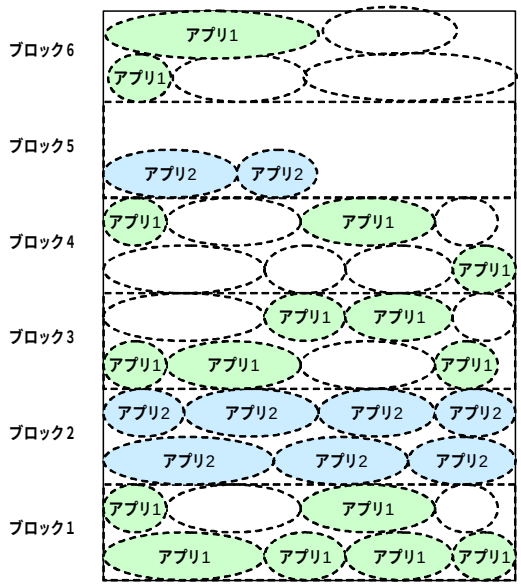

図 6: **アプリ** 1 **のスイープ実行後**

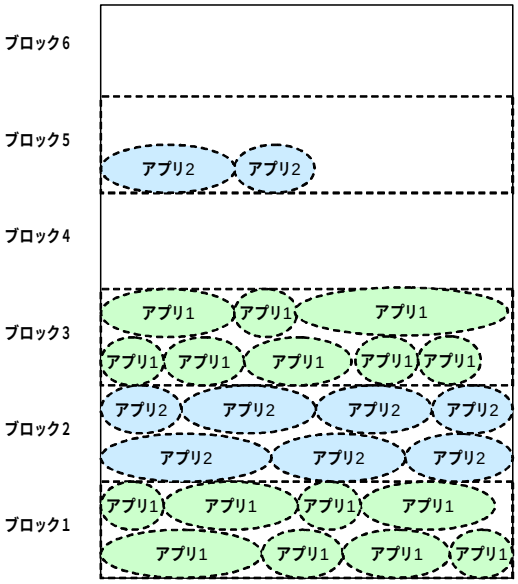

図 7: **アプリ** 1 **のコンパクション終了後**

め、GC 対象としてアプリケーション 1 を選択し、アプリケーション 1 に対して GC を実行するものとする。

まずマーク工程で、アプリケーション 1 の必要データにマークを付ける。これによりアプリケーション 1 の不要データはマーク付けされていない状態となる。

次にスイープ工程で、アプリケーション 1 の不要データを回収する。ブロック管理テーブルの情報により、アプリケーション 1 の不要データはブロック 1、3、4、6 内だけに存在することを特定し、ブロック 2、5 をスイープ対象外とする。これにより、アプリケーション 1 の不要データを回収し、空き領域とすることができる(図 6)。

さらにコンパクション工程で、アプリケーション 1 の必要データを再配置する。再配置先はアプリケーション 1 に割り当て済みのブロック 1、3、4、6 である。この結果、アプリケーション 1 のデータはブロック 1、3 に再配置され、ブロック 4、6 は空き領域となる(図 7)。

最後に JavaVM はブロック 4、6 が空き領域になったことをブロック管理テーブルに登録する。以上により、アプリケーション 2 の状態に関係なく、アプリケーション 1 に対する GC の実行を完了できる。

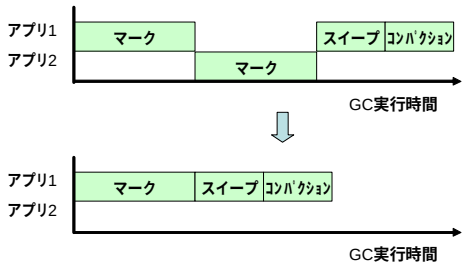

図 8: GC **実行時間の短縮**

以上により、図 8に示す通り、1 回の GC 実行時間を短縮できる。

3.5. チューニングパラメタ

メモリブロック化方式では、アプリケーションの実行性能、メモリの利用効率、及び GC の実行性能に対して、以下のパラメタが影響を及ぼすと考える。

a) GC の実行条件
b) GC 対象アプリケーションの選択条件
c) メモリブロックの割り当て

3.5.1. GC の実行条件

複数のアプリケーションが動作する環境では、GC の実行が他のアプリケーションに影響を与える可能性がある。従って、GC を実行する条件を設定し、GC の実行による影響を最小限に抑える必要がある。

GC の実行条件の例を以下に示す。

a) メモリブロックの枯渇
実行中のアプリケーションのメモリ確保や新たなアプリケーションの起動により未使用ブロックが減少する。
GC 実行条件とする未使用ブロック数を閾値として予め設定する。

b) アプリケーションの状態遷移
複数のアプリケーションが動作する環境ではアプリケーションの状態が変化する。例えば、実行状態から停止状態への遷移や、前面から背面への移動等がある。停止状態や背面のアプリケーションに多くのリソースを割り当て放置することは非効率的であるため、GC の実行条件として設定する。

3.5.2. GC 対象アプリケーションの選択条件

複数のアプリケーションが動作する環境では、GC の対象とするアプリケーションを選択する必要がある。

GC 対象アプリケーションの選択条件の例を以下に示す。

a) 停止状態に遷移するアプリケーション
b) 停止状態のアプリケーション(新たなアプリケーションを起動する際にメモリが充分にある場合、他のアプリケーションの GC を実行せず、直ちに停止状態にした方が起動時間を短縮できる。このため、GC 未実施で停止状態のアプリケーションが存在する可能性がある。)
c) 割り当て済みのブロック数が多いアプリケーション

3.5.3. メモリブロックの割り当て

メモリブロック化方式では、メモリブロックの割り当てに関して、以下の 2 つのパラメタを決定する必要がある。

a) ブロックサイズ
ブロックサイズが過大の場合、ブロック内の未使用領域が無駄になる可能性がある。
ブロックサイズが過小の場合、ブロックの割当回数の増加やブロック内で要求された空き領域を確保できないことにより、未使用領域が発生する。
これらはメモリの利用効率を低下させる。

b) ブロックの割り当て方法
短時間に大量のメモリを必要とするアプリケーションではブロック割り当てのオーバヘッドが無視できなくなる。
この対策の一例として、一定時間内に M 個以上のブロックを要求した場合、1 回の要求で N 個のブロックを割り当てる等、ブロックの割り当てを増加させ、ブロック割り当てのオーバヘッドを抑える方法がある。
なお、携帯端末では同時に動作するアプリケーション数は高々10 程度であり、ブロックを分割することによるオーバヘットを考慮し、ブロック数は最大でも 100 個程度が適切と考える。

4. 評価

本章ではメモリブロック化方式によるアプリケーションの実行に対するオーバヘッドを考察し、定性的に評価する。

メモリブロック化方式によるオーバヘッドとしては以下の項目が挙げられる。

a) メモリ割り当て
b) メモリアクセス
c) GC 性能

4.1. メモリ割り当て

メモリブロック化方式はメモリをブロックに分割し、アプリケーションの要求に応じて動的にメモリを割り当てる。このため、アプリケーションのメモリ利用状況に応じた柔軟なメモリ割り当てが可能である。

アプリケーション実行中のメモリ割り当ての効率に関しては、要求したメモリのサイズ以上の空き領域が割り当て済みブロック内に存在する場合、ブロックからメモリを割り当てるためオーバヘッドは発生しない。

ブロックに充分な空き領域が存在しない場合、JavaVM がアプリケーションにブロックを割り当てる際のオーバヘッドが生じる。これにはブロック管理テーブルへの登録処理も含まれる。

アプリケーションへのブロック割り当てのオーバヘッドを削減する方法として、アプリケーションに予め充分なブロックを割り当て、ブロック割り当ての回数を削減する方

法がある。しかし、これはメモリの利用効率とのトレードオフとなる。

4.2. メモリアクセス

アプリケーション実行中のメモリアクセスに関しては、アプリケーションに割り当てられたブロック内のデータを直接参照するため、メモリブロック化によるオーバヘッドは生じない。

4.3. GC 性能

GC 性能は 1 回の実行時間とスループットで評価する。

a) 1 回の実行時間

メモリブロック化方式は、アプリケーション単位で GC を実行可能である。このため、全てのアプリケーションに対して GC を実行する場合と比較して、1 回の実行時間を短縮できる。

また、GC 対象として停止中のアプリケーションを選択した場合、前面で実行中のアプリケーションに影響が出ないように GC の実行優先度を下げて、断続的にマーク工程を実行する等の柔軟な対応が可能である。

b) スループット

従来のマーク&スイープ方式とメモリブロック化方式の違いは、メモリブロック化方式ではスイープ工程においてブロック管理テーブルを参照し、スイープ対象とするブロックを選択する処理である。

ブロック管理テーブルを参照する処理が追加されるが、このオーバヘッドは軽微である。また、スイープ対象ブロックを限定するため、不要データの検索範囲が縮小する効果がある。

5. おわりに

本論文では単一の JavaVM 上で複数の Java アプリケーションを実行する際のメモリ割り当てと GC 方式として、メモリブロック化方式を提案した。

メモリブロック化方式はメモリを複数のブロックに分割し、アプリケーションにブロック単位でメモリ割り当てを行うことにより、アプリケーション単位の GC 実行を可能にする。これにより 1 回の GC の実行時間を短縮できる。また、メモリブロック化方式のオーバヘッドによるアプリケーション実行性能に与える影響は軽微であると考える。

今後は本方式を実装し、チューニングパラメタを変化させて、アプリケーションを動作させ、性能評価を実施する予定である。

参考文献

[1] J2SE 1.5 Class Data Sharing, http://java.sun.com/j2se/1.5.0/docs/guide/vm/class-data-sharing.html

[2] Grzegorz Czajkowski, Laurent Daynes, Multitasking without Compromise: A Virtual Machine Evolution, ACM OOPSLA (2001)

[3] Java 2 Platform Standard Edition, http://java.sun.com/j2se/1.5.0/

[4] JSR-121 Application Isolation API Specification

[5] Paul R. Wilson, Uniprocessor Garbage Collection Technique, International Workshop on Memory Management (IWMM'92), (Sept. 1992)

[6] http://jp.sun.com/products/software/consumer-embedded/kvm/

[7] John McCarthy. Recursive Functions of Symbolic Expressions and Their Computations by Machine, Communications of the ACM, Vol.3, No.4, pp.184-185, (Apr. 1960)

[8] CLDC HotSpot Implementation Virtual Machine, http://java.sun.com/products/cldc/wp/CLDC-HI_whitepaper-March_2004.pdf

[9] Henry Lieberman and Carl Hewitt.: A Real-Time Garbage Collector Based on the Lifetimes of Objects, Communications of the ACM, Vol.26, No.6, pp.419-429, (Jun. 1983)

[10] George E. Collins. A Method for Overlapping and Erasure of Lists, Communications of the ACM, Vol.2, No.12, pp.655-657, (Dec. 1960)