修 士 論 文

リアルタイム環境に適した Garbage Collection API の研究

指導教官　権藤 克彦 助教授

北陸先端科学技術大学院大学
情報科学研究科ソフトウェア基礎講座

八十島 利典

平成12年 2月 15日

要 旨

本研究では、リアルタイム環境に適した Garbage Collection API を提案、実装および検証する。本研究で用いるプログラミング言語 Java は高い移植性と再利用性など良い性質をもつことから広く利用されているが、リアルタイムシステムにおいてはソフトウェアの保守性を保ちつつ、システムが要求する性能を満たすことは難しい。なぜなら、GC を持つシステムでは、最大応答時間の保証、GC の処理時間の予測、ヒープ残量の保証とメモリ割り当てに必要な時間の予測が一般に難しいからである。

また、GC の性能は使用するアプリケーション、動作環境やパラメータによっても大きく変化する。つまり、アルゴリズムの改良や GC のチューニングだけでは大きな性能向上は難しい。

よって、本論文ではリアルタイム処理に適した GC の機能として Java のガベージコレクション API(GC API) を提案する。この GC API を用いると、プログラマはリアルタイム制約を満たすためにアプリケーションごとに GC の動作を変更可能となる。我々が提案する GC API は、1)GC の実行時間や優先順位の指定、2)GC の起動抑制・許可、3) メモリ予約である。もちろん、これ以外にも多くの有用な API があるが、実現と実験を優先すべきと考えて、あえてシンプルな API とした。

これらの GC API を用いることで、アプリケーションごとに適切に GC のパラメータ調整できればリアルタイム制約の充足が期待できる。

残念ながら、デバックやチューニングが不十分なため、GC API を用いて有意な差がでるプログラムを動作できていないが、GC の概念や基本的なアルゴリズム、リアルタイムシステムの GC に対する要求や設計および実装段階で得られたことについて説明する。

目 次

図 目 次

表 目 次

第 1 章

はじめに

1.1 背景

近年、プログラミング言語 Java は広く利用されている。なぜなら、Java は高い移植性と再利用性など良い性質を持つからである。Java の高い移植性と再利用性を使用するために、組み込みシステムやリアルタイムシステムにまで応用範囲が広がりつつある。

リアルタイムシステムで Java を用いるために、Java のリアルタイム拡張仕様 [6] [7] の作成が進められているが、ソフトウェアの保守性を保ちつつ、これらのシステムが要求する性能 (例えばリアルタイム性) を幅広く満たす事は難しく、まだ実現には至っていない。

システムが要求する性能を満たすことが難しいにもかかわらず、リアルタイムシステムにおいても Java を利用する理由として高い移植性と再利用性があり、それにはメモリ管理をガベージコレクション (以下 GC) に任せられることが大きく貢献している。すなわち、リアルタイムシステムにおいても、Java がリアルタイム性を満たすことが可能となれば、複雑な製造工程の制御やロボットなどコンピュータ制御に依存している複雑なシステムにおいても Java の高い移植性と再利用性が利用可能となる。

しかし、GC はリアルタイム制約を満たしにくくする。なぜなら、GC を持つシステムでは、最大応答時間の保証、GC の処理時間の予測、ヒープ残量の保証とメモリ割り当てに必要な時間の予測などが一般に難しいからである。リアルタイム GC に要求されることは以下の 3 つである。

(1) リアルタイム性を満たすこと

(2) メモリを滞りなく供給すること

(3) コードの理解がしやすく、保守しやすいこと

しかし、リアルタイム GC への要求を満たすためには解決すべき問題が上記の要求に対応して３つある。

(1) リアルタイム性を満たすためには、GC による停止時間をできる限り小さなものにしなければならない。なぜなら、GC によって生じる停止時間によってアプリケーションがデッドラインをミスする可能性があるからである。

(2) GC によって生じる停止時間を減少させるために GC の実行頻度を減らすと、メモリ供給が滞る可能性がある。つまり、メモリの管理・供給は GC が行っているからである。

(3) GC を使用しなければ、ユーザによる明示的なメモリ管理が必要となるが、保守のためのコストが非常に高い。すなわち、アプリケーションプログラム内にメモリ管理用のコードが散らばり、コードの追跡が困難となる。また、Java の高い移植性と再利用性およびソフトウェアの保守性が損なわれる。

また、GC の性能は使用するアプリケーション、動作環境や GC のパラメータによっても大きく変化する。つまり、アルゴリズムの改良および GC のチューニングによってリアルタイム制約の充足を計ろうとすれば、使用するアプリケーションや動作環境に合わせた改良を行わなければならない。これもまたソフトウェアの保守性が損なわれることになる。また、既にリアルタイム GC アルゴリズムの研究は多く行われており、アルゴリズムの改良による大きな性能向上は難しい。

ところが、パラメータ変更による GC の性能向上の体系的な研究は、あまり行われていない。つまり、使用するアプリケーションや動作環境に合わせて GC のパラメータ調整を行うことが可能であれば、使用環境に特化した GC として動作させることが可能となる。よって、GC のパラメータ調整によるリアルタイム制約の充足が期待できる。

つまり、ここでの問題は以下である。

- GC によりソフトウェアの保守性は保たれるが、リアルタイムシステムの要求する性能を幅広く満たすことは困難
- GC は Java のリアルタイム拡張においてリアルタイム制約を満たしにくくすることからチューニングが必要

上記の問題を解決することが可能であれば、リアルタイム環境においても Java の高い移植性と再利用性を利用可能であると考える。

1.2 目的

本論文では、リアルタイム処理[1]に適した GC の機能として、Java のガベージコレクション API(GC API) を提案する。この GC API を用いると、プログラマはリアルタイム制約を満たすためにアプリケーション毎に GC の動作を変更することが可能となる。本来は、GC はアプリケーションプログラマに対して透明、つまりメモリ管理を意識させないことが重要であるが、我々は次の理由から GC API が有用であると主張する。

- リアルタイム用の GC アルゴリズムの改良や GC のチューニングだけではリアルタイム制約の充足は難しい。GC の性能はアプリケーション、動作環境、GC のパラメータによっても大きく変化するからである。
- アプリケーションや動作環境ごとに自動的に GC の動作を調整できれば理想的だが、現在の技術では難しい。
- 多くの場合、アプリケーション開発者はどのように最適化すれば良いかを知っている。また、最適化が必要な部分はコードの一部にすぎないことも多い。それゆえ、GC API があれば、保守性を保ったまま、そのアプリケーションに特化した GC の動作のチューニングをしやすい。

我々が提案する GC API は、次の 3 種類である。

- GC の実行間隔や優先順位の指定

  Timed-GC は一定間隔の実行間隔で動作する。実行間隔が短ければメモリ回収量は多くなるが、overhead も大きくなる。実行間隔が長くなればその逆となる。アプリケーションによって、適切な実行間隔は異なると考えられることから、各アプリケーションに適した実行間隔を指定する。

- GC の起動抑制・許可

  アプリケーションの種類によっては実行時に時間的制約の非常に厳しい期間が存在する。この期間内に GC の割り込みが発生するとアプリケーションがデッドラインミスする可能性がある。よって、時間的制約の厳しい期間では GC を抑制することで、デッドラインミスの危険性を回避することが可能となる。

- メモリ予約

[1]本論文ではソフトリアルタイム処理 (リアルタイム制約の充足を完全には保証しない) を仮定する。

GC を持つ実行環境では、メモリ割り当て時に、GC を少し実行させたり、GC に必要な処理 (割り当てたメモリのマーク領域の確保と初期化など) を行うため、一般に overhead がある。また、ヒープ残量不足で、GC の終了待ちが必要なこともある。つまり、ヒープ残量の保証やメモリ割り当てに必要な時間の予測が難しい。このため GC を経由せずメモリをあらかじめ割り当てておきたいことがある。この機能をメモリ予約と呼ぶ。メモリ予約の API は、GC を経由せずに明示的に動的メモリ管理をさせることで、メモリ残量を保証し、メモリ割り当てに必要な時間をより小さく、より予測可能にする。

もちろん、これ以外にも多くの有用な API があるが、我々は実現と実験を優先すべきと考えて、あえてシンプルな API とした。

これらの GC API を用いることで、アプリケーション毎に適切に GC のパラメータを調整できれば、さらなる停止時間の減少が期待できる。よって、本方式の有効性を確認するため実験的に実装し、リアルタイム GC のリアルタイム制約の充足を試みる。また、アプリケーション毎の適切なパラメータを取得するためにサンプルプログラムを用いることで実験的に適切なパラメータを得る。

また本研究で採用する Timed-GC はインクリメンタルおよび時間によるトリガを兼ね備えた GC である。インクリメンタル GC とは、GC のマーキングの最中であってもユーザのプログラムが動作可能であり、これによって GC の停止時間を短くすることが可能である。また、時間によるトリガは一定間隔で GC が実行されることから、周期的なアプリケーションのメモリ消費の割合とコレクションの割合を調整することが可能であり、非周期的なアプリケーションであったとしても一定間隔の GC の実行によりメモリ枯渇の危険性が少なくなる。また、全体的な overhead は大きくなるが、１回あたりの overhaed が小さくなることで GC による停止時間を減少することが期待できる。よって、上記の理由から Timed-GC は GC の停止時間の減少が期待できる。

1.3 実現の概要

本研究において Kaffe-1.0.b1-rt[8] というリアルタイム化された Java 言語システムを使用した。しかし、GC は基本的な mark-sweep アルゴリズムであるので GC による停止時間がありリアルタイム制約は満たしにくい。よって、Kaffe においてリアルタイム制約を満たすために Timed-GC および GC API を実現した。

まず、Timed-GC を実現するために以下の修正を加えた。

- 湯浅のスナップショットアルゴリズム [2][11] を用いて、kaffe-1.0.b1 の GC を並行 GC とした
- write-barrier はメモリ保護機構と RT-Mach の例外処理機構を用いた OS 支援方式で簡易に実現した
- ヒープの残量だけでなく、一定時間毎に GC を動作させるためにタイマスレッドを追加した

次に、GC API を実現するために以下の修正を加えた。

- GC の実行時間や優先順位の指定

  実行時間を変更するために、タイマスレッドに変更を加えた

  優先順位を変更するために、kaffe_thread_interface を用いて GC のスレッド ID を取得し設定可能に変更を加えた

- GC の起動抑制・許可

  GC スレッドを停止させることで GC の起動抑制・許可の指定を簡易に実現した

- メモリ予約

  メモリ予約は設計段階を終了しており、実装することで実現可能と考えている

1.4 本論文の構成

本論文の以降の構成を以下に示す。

2章 一般的な GC の手法 (本研究で使用した Timed-GC で用いられている mark-sweep 方式も含む) を説明する。また、ポインタとデータの区別できない環境では、ポインタとして合法な値は全てポインタとして処理する conservative GC の説明と、mark-sweep 方式で生じる断片化を部分的に解決する two-level allocation について説明する。

3章 リアルタイム環境で GC を使用するために一般的に用いられる手法である incremental GC の説明と、ユーザプログラムの割り込みによって生じる問題を解決する barrier methods を説明する。

4章 本研究で導入するGC APIを提案する、GCの起動抑制・許可、GCの実行間隔や優先順位の指定およびメモリ予約について説明する。

5章 関連研究として、RTJEG、J consortiumとCMMによって提案されているAPIの説明を行う。それぞれの研究において独自の提案がなされており、それらの違いについても説明する。

6章 GC APIの実装対象であるkaffe-1.0.b1-rtの概要を説明し、GC APIの実現のために用いたTimed-GCの実現方法を説明する。また、GC APIを用いたプログラム例を説明する。

7章 最後にこの論文についてのまとめを行い、今回の研究で得られたこと、今回の研究で達成できなかったことなどについて述べる。また、これからの展望について説明する。

第 2 章

基本的な Garbage Collection

この章では、まず始めに GC に関する一般的な概念や用語についての説明を行う。また、GC の基本的な手法 (reference count、mark-sweep、copying)[2] の説明と、本研究で用いている GC の手法 (conservative GC、two-level allocation)[2] についての説明をする。

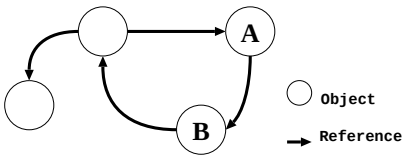
2.1 Garbage Collection とは

動的なデータ構造

一般にプログラミング言語では、ヒープ上に動的にデータを生成する。動的に生成されたデータを、この論文ではオブジェクトと呼ぶ。

あるオブジェクトが生成されると、そのオブジェクトへの参照がオブジェクトの生成要求元に返される。その参照は、大域変数やオブジェクトの内部変数として保持されたりする。このオブジェクトと参照の関係は有向グラフとして表すことができる (図 2.1)。

図 2.1 において、オブジェクト A はオブジェクト B を参照している。このとき、オブジェクト A からオブジェクト B への参照方向を持つ辺として有向グラフが表される。



ルート

プログラムは、大域変数やスタックからなる参照ポインタの集合を持つ。このようなヒープデータへの参照を保持している場所は処理の根となる。つまり、そのような場所のことをルートと呼ぶ。プログラムはルートから参照パスを経由してオブジェクトを参照する。ルートからたどることができるオブジェクトは reachable と呼ばれ、たどれないものを unreachable と呼ぶ。一度 unreachable となったオブジェクトは、再びプログラムから利用されることは無い (図 2.2)。

図 2.2 において、白いオブジェクトはルートから参照することができないことから unreachable である。

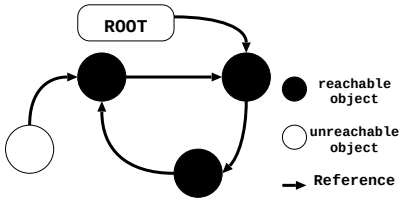


図 2.2: **reachable** と **unreachable** なオブジェクト

ミューテータ (mutator)

プログラムによるオブジェクトの生成や破棄などによって、オブジェクトと参照の関係を表す有向グラフは変化 (mutate) する。このことを mutation と呼び、変化を引き起こすプログラムの計算プロセスを mutator と呼ぶ。

ゴミ (garbage)

オブジェクトと参照の関係を表す有向グラフは mutator による変化によって、ルートから参照されないオブジェクトを生み出すことがある。そのオブジェクトは unreachable であり、再びプログラムから利用されることは無い。このような unreachable なオブジェクトのことをゴミと呼ぶ。図 2.2 の unreachable オブジェクトはゴミである。

ガベージコレクション (garbage collection)

ゴミとなったオブジェクトであってもメモリが割り当てられている。プログラムの実行によりオブジェクトを生成し続ければ、メモリは有限なので、いつかはメモリはゴミによって埋め尽くされる。すなわち、ゴミを回収して不要なメモリを回収する必要がある。

ゴミを回収するには２つの方法がある。

(1) プログラマが明示的にゴミを回収する

(2) 言語処理系などのシステムが自動的にゴミを回収する

しかし、(1) の方法だと保守のためのコストが非常に高くなってしまう C 言語でいえばメモリを malloc によって確保し、free によって開放する訳だが、free を忘れると使用済のメモリが増え、また、free してはいけないものを free してしまうと、予期せぬ場所でエラーが発生する原因となる。ゆえに、デバック等の保守が困難になる。このことから、近年では (2) の方法が主に用いられている。この (2) を行う機能をガベージコレクションという。

2.2 Reference counting 方式

Reference counting アルゴリズムは、あるオブジェクトから各々のオブジェクトへの参照の数を数えることをベースとしている。この利点はオブジェクトが使用されているかどうかの単純な追跡にある。

各々のオブジェクトには参照数を数えるための領域 *reference count* を持つ。メモリマネージャは、オブジェクトへの参照数を正確に管理しなくてはならない。

未使用のオブジェクトはゼロの参照数を持つ。新しいオブジェクトがメモリのプールから割り当てられた時、参照数は１へセットされる。このオブジェクトへの参照が１つ増えるたびに、参照数は１増加される。また、このオブジェクトへの参照が１つ減るたびに、参照数は１減少される。参照数が０へ落ちた時、そのオブジェクトはどこからも利用されることがなくなるのでゴミとなる。このようにゴミを識別する方法を reference counting 方式と呼ぶ (図 2.3)。

Reference counting 方式の長所

Reference counting 方式の長所は、そのオブジェクトがゴミであるかどうか参照数を見るだけで確認できることである。ゴミと識別されたオブジェクトは直ちに回収されること

ができる。よって、一回の回収のための時間が短くて済む。

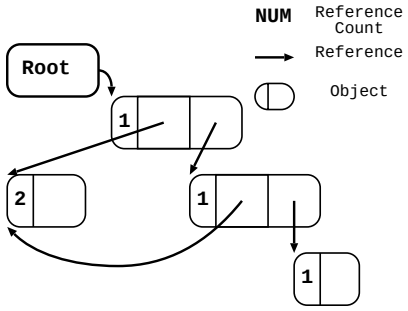


図 2.3: Reference counting 方式

Reference counting 方式の短所

Reference counting 方式の短所は、大きく 3 つある。

(1) 循環データ構造を持つゴミを回収できない

循環データ構造を持つゴミは、ルートからの参照を持たないが参照数が 0 にならないことからゴミとして扱われず、コレクターに回収されない (図 2.4)。この図では、*before* の状態では、他のオブジェクトからの参照がある。しかし、*after* では、ルートからの辿れないが参照数は 0 とならない。よって、ゴミとして扱われず回収されない。

(2) 毎回ポインタがアップデートまたはコピーされるたびに参照数の調節が必要

ポインタがサブルーチンを通過した時でさえ参照数を増加させ、返った時には参照数を減少させなければいけないからである。

(3) 参照数を保持する余分な空間が必要

最悪、この空間はヒープ内とルート内に保持されているポインタの合計数を保持する十分な大きさでなければいけない。

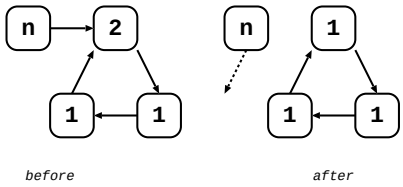


図 2.4: 循環データ構造

2.3 Mark-sweep 方式

Mark-sweep 方式では、オブジェクトがゴミになってもすぐには回収せず、全ての利用できるメモリを使い果たすまで、オブジェクトはゴミのまま残る。メモリを使い果たした状態で、新たなオブジェクトを生成するリクエストが来ると、ガベージコレクタールーチンが使用済のオブジェクトを回収するために呼び出される。その間は、アプリケーションプログラムの実行が中断される。

Mark-sweep は回収できるオブジェクトを特定するために、全てのオブジェクトをトレースする。このトレースはルートから開始され、到達可能なオブジェクトにマークをつけることでゴミかどうかを識別する。

図 2.5 のルートからトレースがはじまり、オブジェクトの *mark-bit* に印をつけてゆくことで、そのオブジェクトがゴミであるかどうかを識別する。Mark-sweep 方式では、ルートから到達することのできない物は全てゴミと考えるので、循環データ構造が存在してもゴミとして処理できる。

もしガベージコレクタが十分なメモリを回収できたならば、ユーザプログラムのリクエストを満足させて、アプリケーションプログラムが再開される。このようにゴミを識別する方法を mark-sweep 方式と呼ぶ。

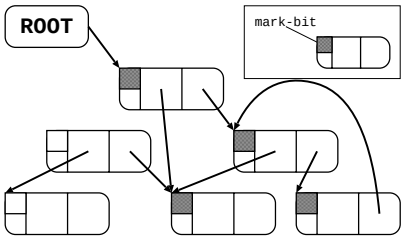


図 2.5: Mark-sweep 方式

Mark-sweep 方式の長所

循環データ構造を特別な処理無しに扱えることが可能である。

Mark-sweep 方式の短所

Mark-sweep 方式の短所は大きく２つある。

(1) stop/go アルゴリズムである

Stop/go は、ガベージコレクターが実行されている間 は計算が停止することである。1980 年代初頭のある大きな Lisp プログラムにおいて、25 %から 40 % を marking と sweeping に費し、ユーザは平均 79 秒毎に 4.5 秒待たされていた。よって、全体を一度に横断する mark-sweep アルゴリズムでは、リアルタイムまたは分散システムには向いていない。

- 単純なアルゴリズムでは memory を断片化する傾向がある

断片化とはオブジェクトがヒープの領域で分散させられることである。断片化によってキャッシュミスが多くなる。これは、不必要な領域を整理することなく再利用しようとするために生じる問題である。

2.4 Copying 方式

Copying 方式では、ヒープの領域を等しく２つの *semi-spaces* に分割する。ひとつは現在のデータを含む領域で、もうひとつは GC の作業領域である。copying ガベージコレクションでは、２つの空間の役割を入れ換えることによって開始される。コレクタはその時、古い semi-space、*Fromspace* 内のアクティブなデータ構造を横断する。オブジェクトに最初に訪れた時、新しい semi-space、*Tospace* 内へオブジェクトをコピーする。

Fromspace 内の全てのアクティブなオブジェクトをトレースした後、アクティブなデータ構造のレプリカを Tospace 内に生成して、ユーザプログラムは再び活動を開始する。使用済のオブジェクトは古い semi-space、Fromspace に放置される。

copying ガベージコレクションの副作用として、アクティブなデータ構造は Tospace の底にぎっしりと詰め込まれる。Compacting コレクタは、断片化問題のあるコレクタより効率的なオブジェクトの割り当てが可能である。このようにゴミを識別する方法を copying 方式と呼ぶ (図 2.6)。

図 2.6 のルートから探索が開始され、まず A を Fromspace から Tospace へコピーする。このとき、A が Tospace のどのアドレスにコピーされたかを示す *forwarding address*(図中の実線) が張られる。単純なコピーだと、コピーされたオブジェクトは元指していたオブジェクトを指す。しかし、その他のオブジェクトからも指されている可能性がある。つまり、forwarding address を用いて、ポインタを張り変える。次に B をコピーし、A から B へ張られているリンクを A' から B' へも張る。これを同様に D まで左再帰で繰り返し、最後にルートのリンクを張りかえる。Fromspace の内容は Tospace へコンパクトに移される。Tospace へコピーされずに残ったオブジェクトはゴミとして回収される。

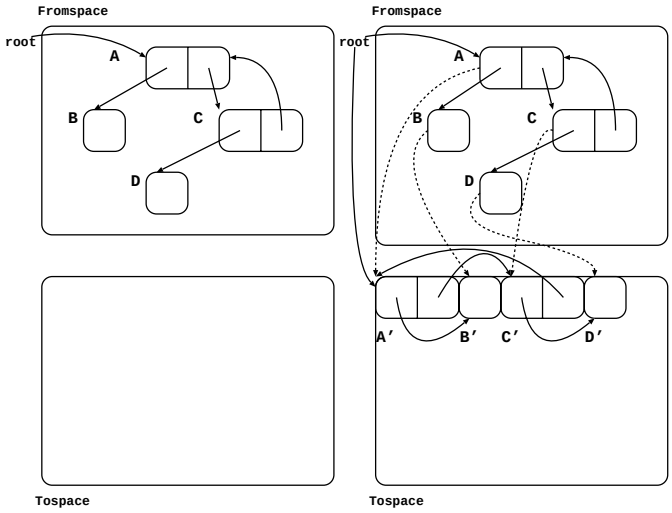


図 2.6: Copying 方式

copying 方式の長所

copying 方式の長所は、アロケーションのコストが極めて低いことである。これは、チェックは単純なポインタの比較であること、新しいメモリはフリースペースのポインタを増加させることによって単純に得られること、そして断片化は Tospace の底からアクティブなデータを詰めてゆくことによって除かれることである。

copying 方式の短所

copying 方式の短所は、２つの semi-spaces を使用することである。要求されるアドレス空間は、non-copying コレクターと比較して２倍である。つまり、他の方式で使用でき

るメモリの半分以下しかメモリを使用することができない。

2.5 Conservative 方式

ポインタとデータの区別ができない場合があり、かつ、その場合にポインタとして合法な値は全てポインタとして扱う GC を conservative GC と呼ぶ。本研究で使用している Kaffe-1.0.b1 の JVM 部分の実装は C 言語であることから、メモリの内容を見ただけでは、どれがポインタであってどれがデータであるかの区別ができない。Lisp などの他言語の処理系においてはデータ自体にポインタとデータの区別があるが C 言語の処理系には存在しない。このことから、本研究では、Conservative 方式を導入している。

しかし、全てをポインタとみなしてしまうと回収すべきゴミを回収し損ねるメモリリークが発生する可能性がある。これは、全てをポインタとみなしていることから、例えば 0x01 番地のデータがゴミであったとしても、どこかのオブジェクトの変数に 1 という数値が入っていたとしたら、誤って 0x01 番地のデータは参照されているとみなされ、このデータは回収されることがない (図 2.7)。

よって、メモリリークが起きること、オブジェクトを別アドレスに移すことができないため copying や compaction ができないという欠点がある。

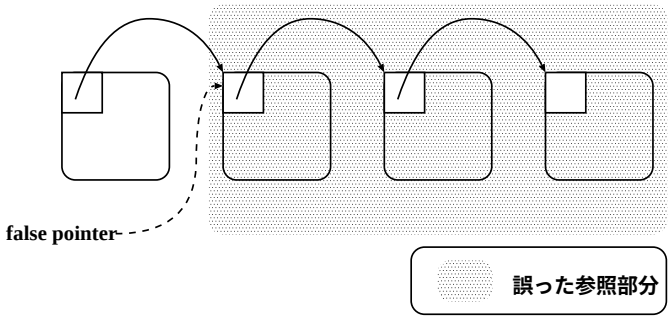


図 2.7: 誤った参照によるメモリリーク

2.6 Two-level allocation

Two-level allocation は、Boehm-Demers-Weiser コレクタ [2] によって使用された。この方法は、conservative GC の欠点であるメモリの断片化を軽減するため、動的なメモリ割り当てを 2 階層にすることをいう。

方法としては、下位レベルで、アロケータはメモリのブロックのリストを保持する。上位レベルで、各々のフリーリストは下位レベルのアロケータから獲得したブロック内部の固定サイズのオブジェクトを割り当てる。

例えば、下位レベルではシステムからまとまった大きさ (例えば 1MB) で動的にメモリを確保し、上位レベルではそれをブロック毎に固定サイズ (例えば 16B、32B、64B...) を決めて,GC にメモリを供給する。

フリーリストが空でなければ、オブジェクトを容易に割り当てることができる。もしガベージコレクションのスイープフェーズで、ブロックが全て空だと確認されたら、異なるフリーリストを回収するために、下位レベルのアロケータへと返ることができる。よって、メモリ割り当てを高速に行うことができる (図 2.8)。

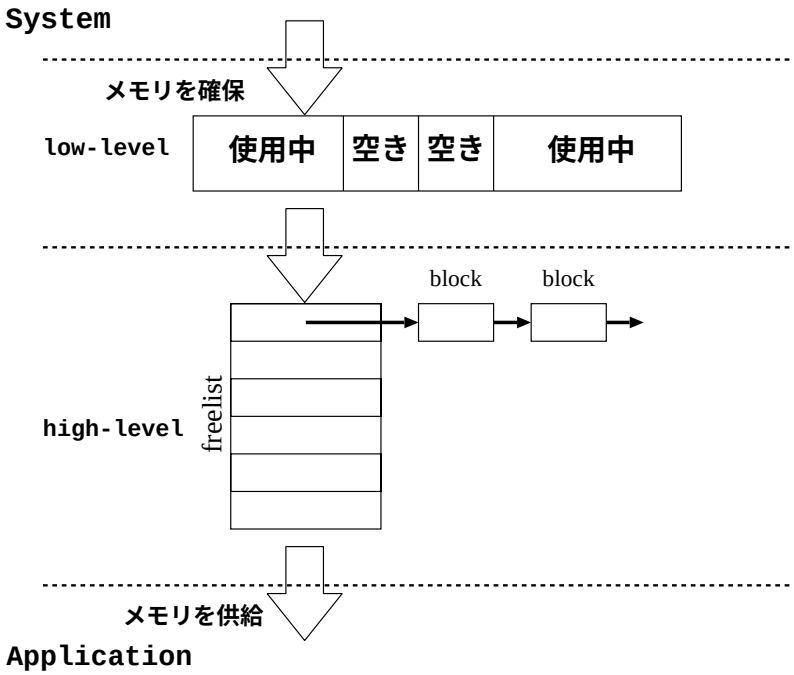


図 2.8: Two-level allocation

第 3 章

リアルタイム環境のための Garbage Collection

この章では、リアルタイム環境で GC を使用するために一般的に用いられる手法である incremental GC についての説明をする。主に、本研究で用いられている kaffe-1.0.b1 の GCmark-sweep をもとにしたものであることから、incremental mark-sweep GC についての説明をする。また、GC の最中に mutator の割り込みによって GC の作業グラフに変更が起こると、参照されているオブジェクトがゴミとして扱われる可能性がある。よって、incremenatl GC 中のグラフの変更を阻止する barrier methods の説明をする。そして、その実現方法の説明をする。

3.1 Incremental GC

リアルタイム環境においては、ガベージコレクションによる停止時間をいかに短くするかが主な問題となる。逐次型のコレクションが招く停止を避けるために多くの研究者が incremental GC を研究してきた。

しかし、逐次ガベージコレクションであっても、コレクションをユーザプログラムの実行とインターリーブすることによってインクリメンタルにすることができる。つまり、従来のガベージコレクションであればコレクションが終了するまでユーザプログラムは停止してしまう。しかし、一回のコレクションの量を減らし、ユーザプログラムとコレクションを交互に実行することで停止時間を短くすることが可能である。このような GC の手法を incremental GC と呼ぶ。

ところが、コレクションサイクルが終了する前に、ユーザプログラムがメモリを使い果

たさないように、コレクタが十分に進むように注意を払わなければならない。これを行う方法のひとつは、コレクションの割合をメモリの消費の割合に調節することである。この方法は、各メモリ割り当て時に小量のマーキングやコピーが行える。

コレクションサイクル終了前にユーザプログラムがメモリを使い果たすこと無く、これを行うためには、非インクリメンタルなものと比較してヒープ中に余分な領域を必要とする。

Incremental mark-sweep GC

Incremental mark-sweep GC は、mutator を実行しながらマークする手法である。基本的なアルゴリズムは mark-sweep と同じである。しかし、マーキングをユーザプログラムの実行にインターリーブしながら行うことで、基本的な mark-sweep による停止時間を抑えることが可能である。

同期

Incremental mark-sweep GC において、ミューテータとコレクタの非同期の実行は一貫性の問題を引き起こす可能性がある。図 3.1 が示すオブジェクトと参照の関係があるとする。ステップ 1 で、コレクタが A を訪問し、ステップ 2 で B を訪問する。しかし、その訪問の間に C への参照が張り変えられたとすると、C は誤ってゴミとして回収されてしまう。つまり、データ構造の結合が変更されたことを指摘するために、ミューテータとコレクタの間には同期が必要である。

Tricolour marking

Incremental GC を記述するためにマーキングの色を 3 色 (tricolour) にする抽象化を用いる。この抽象化でヒープ中のオブジェクトは以下のどれかに塗られる。

Black そのオブジェクトと直下の子供達が訪問されたことを示す。コレクタは黒いオブジェクトに対しては終了しており、再び訪れる必要は無い。

Grey そのオブジェクトはコレクタによって訪問されなければいけないことを示す。灰色のオブジェクトは次の 2 つのどちらかを意味する。しかし、そのどちらの場合も、コレクタは再び訪問する必要がある。

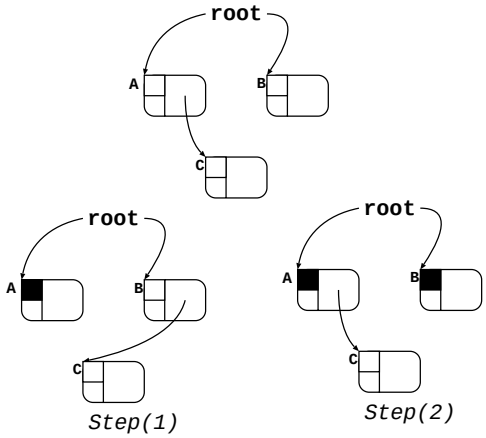


図 3.1: ミューテータとコレクタの非同期による干渉

(1) コレクタによって訪問されたが、その構成ポインタはまだスキャンされていない

(2) コレクタの背後でミューテータがグラフの残りを変更した

White ノードは訪問されていない。マーキング終了後にはゴミを意味する

GC のサイクルは到達可能なノードが全て黒色のときに停止する。

Barrier methods

ミューテータが白へのポインタを黒いオブジェクトに書き込むことによってガベージコレクションが崩壊するのを防ぐには２つの方法がある。

1. ミューテータが白いオブジェクトを決して見ないことを保証する

   白いオブジェクトにアクセスが試みられると、そのオブジェクトは即座にコレクタによって訪問される。これが起こることを保証するために「リードバリア」により白いオブジェクトを保護する。

2. ミューテータが白へのポインタを黒いオブジェクトに書き込んだ場所を記録しておき、コレクタが問題のノードに再訪問できるようにする

   オブジェクトを「ライトバリア」によって保護する。

Incremental mark-sweep GC はこの「ライトバリア」を用いることで、ミューテータが白へのポインタを黒いオブジェクトに書き込む、つまり「黒→白」の参照がおこらないように処置を行っている。ライトバリアの詳しい説明を 3.2 節でする。

Generational GC

Generational GC はincremental GC とは異なる方法で、停止時間を抑える。Generational GC では、最もメモリが回復しやすいヒープ領域に集中してメモリ回収を行う。この領域が比較的小さく、かつオブジェクトの生存率が十分に低い場合は Generational の手法が停止時間を抑えることに有効である。

この手法は、一般的に 2 つ以上の世代 (generation) にヒープを分割する。一度以上使用された新しい世代のオブジェクトは、古い世代のオブジェクト領域へ移される。なぜなら、新しい世代のオブジェクトは、古い世代のオブジェクトよりもすぐに使用されなくなる可能性が高いからである。つまり、古い世代のオブジェクトの回収する回数を減らすことが可能となり、停止時間を抑えることができる。

しかし、古い世代と若い世代の間をまたがるポインタは、前処理によってルート集合の一部分として扱わなければならない。なぜなら、図 3.2 で灰色のオブジェクトは、若い世代のオブジェクトを参照しており、もしこのオブジェクトをルート集合の一部分として扱わなければ、若い世代の領域を GC するときに、誤って回収されてしまう可能性がある。若い世代から古い世代でオブジェクトを参照しているときも同様である。よって、これらの古い世代と若い世代の間をまたがるポインタの更新や参照を捕えるために、バリアが必要となる。

また、新しい世代の領域が大きくなり、オブジェクトの生存率が高い場合には、この手法では停止時間を抑えることはできない。

3.2 Write-barrier とは

ライトバリア (write-barrier) の役割は、GC の作業用グラフの変更がコレクタの走査に干渉することを防ぐことである。図 3.3 において、ミューテータはポインタ A を上書きした。B はどこからか参照されている可能性があるが、このまま放置しておくと B は黒にマークされなかったことからゴミとして扱われ、回収されてしまう可能性がある。つまり、この誤ったメモリの回収を防ぐ役割がライトバリアである。

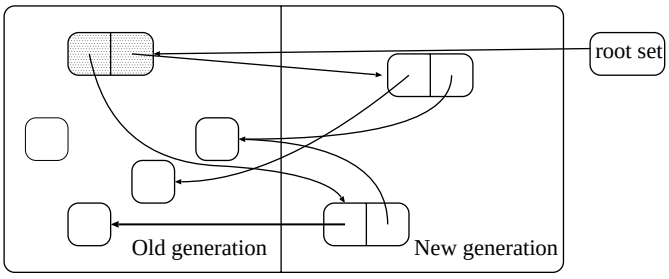


図 3.2: Generational 方式

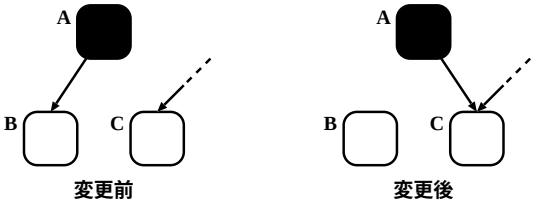


図 3.3: Mutator が B ではなく C を指すような A の更新

3.2.1 Yuasa's snapshot write-barrier

湯淺のスナップショット・ライトバリア (Yuasa's snapshot write-barrier) は、最も良く知られた snapshot-at-the-beginning アルゴリズムである。Snapshot-at-the-beginning アルゴリズムとは、ポインタが上書きされるたびに、元の参照していたオブジェクトは灰色に塗られてマークスタックから指される。灰色に塗られたオブジェクトは再びコレクタによって訪れられることから、白いオブジェクトへの元々の参照の損失を防ぐアルゴリズムである (図 3.4)。

湯淺のライトバリアは、マーキングフェーズ中のポインタの更新を監視し、オブジェクトのマークビットをセットすることで、古い白いポインタを灰色に塗り、マーキングスタック上に古いセルへの参照をプッシュする。

図 3.5 のアルゴリズムで、Updata された先のオブジェクトは shade(P) によってマークビットを marked にされ、マークスタックへとプッシュされる。その後、ポインタ A が指している先を B から C へと変えられる。

この方法はオブジェクトがゴミであるかどうかに関わらず、オブジェクトを保存する。つまり、非常に保守的な方法であるといえる。

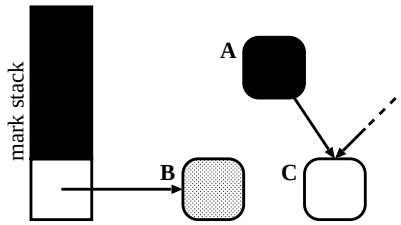


図 3.4: 湯淺のスナップショット・ライトバリア

```
shade(P) =
    if not marked(P)
        mark_bit(P) = marked
            gcpush(P, mark_stack)

Updata(A, C) =
    if phase == mark_phase
        shade(*A)
    *A = C
```

図 3.5: 湯淺のスナップショット・ライトバリア アルゴリズム

3.2.2 Dijkstra's write-barrier

Dijkstra のライトバリアは、親オブジェクトの色を考えずに参照が作られた時に白いオブジェクトを灰色に色を塗る。この方法は保守的なアルゴリズムである (図 3.6 、図 3.7)。

このアルゴリズムでは、各オブジェクトの中に明示的な色ビットを持つ。ターゲットがそのまま黒に色を塗ることはできない。もし黒に塗ってしまうと、「黒→白」参照が発生し、白のオブジェクトはゴミとして誤って回収されてしまう可能性がある。

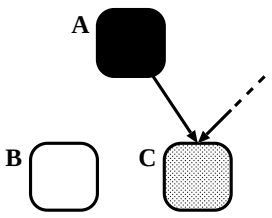


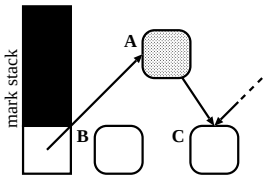
図 3.6: Dijkstra のライトバリア

```
shade(P) =
    if white(P)
        colour(P) = grey
Updata(A, C) =
    *A = C
    shade(C)
```

図 3.7: Dijkstra のライトバリア アルゴリズム

3.2.3 Steele's write-barrier

Steele のアルゴリズムは、白いオブジェクトの黒い親オブジェクトを灰色に塗るアルゴリズムである。このテクニックは、A への余分な訪問を必要とするが、コレクションサイクルの最後で到達不能なゴミの総量は減る。また、湯浅のアルゴリズムと同様にマークビットと灰色参照のマークスタックを使用する (図 3.8 、図 3.9)。



3.3 Write-barrier の実現法

ライトバリアの実現方法にはいくつかの方法がある。

Software barrier

Software barrier は、ポインタの値を変更する全てのコード (例えば `*p1 = p2;`) にバリア処理コードを埋め込むことによって実現される。しかし、ポインタ値を変更する全てのコードにバリア処理が埋め込まれることから、保守のためのコストが非常に高い。

Hardware barrier

Hardware barrier は、付加的な命令を要求しない。特に非協調的なコンパイラを用いる時は有利である。また、overhead が小さいことから Hardware barrier は最も速い。しかし、このバリアを組み込むためのコストは非常に高い。

Operating System support barrier

Operating System support barrier は、OS の仮想メモリ保護機構を使用する。これは、ポインタの領域が更新されると、保護されたページへのアクセスを捕獲するか、もしくはオブジェクトの位置を記録することによって実現される。この方法は、コンパイラへの変更が必要無いことから扱いやすい。 しかし、パフォーマンスは software barrier よりも劣っている。

3.4 Timed-GC

Timed-GC は、Ito ら [1] によって提案されたリアルタイム用の GC である。これは、snapshot write barrier を使用した mark-sweep 方式によって実現されている。

一般的な incremental GC では、メモリの残量が少なくなったときに GC が起動される。しかし、メモリの残量が少なくなるときはアプリケーションプログラムの処理も忙しく、この期間に GC が多く割り込まれるとデッドラインミスを引き起こす可能性がある。つまり、メモリの使用量が少ないときでも GC をすることが望まれる。ところが、メモリの使用量が少ない期間に GC を起動させるためには、GC の起動を明示的に指定する必要がある。なぜなら、メモリの使用量が少ない期間はアプリケーションプログラムの実装に依存するからである。

Timed-GC では一定間隔に GC を起動することで仕事量の少ないときにもメモリの回収が行われる。また、周期的なアプリケーションプログラムでは、Timed-GC であればメモリの使用周期に合わせることが容易である。この観察から Timed-GC は優れていると考えられる。Timed-GC の長所は大きく分けて２つある。

- Incremental である

  単純な mark-sweep アルゴリズムであれば、GC による停止時間によってリアルタイムの反応性を損ねてしまう。しかし、*timer* によって供給されるタイミングシグナルによって、GC の実行を分割することにより、一回の GC による停止時間を短くできる。

- Time-sensitive である

  リアルタイムアプリケーションのリアルタイム要求に合わせてタイマのタイマ間隔を決定することができることから、アプリケーションからのリアルタイムの反応性を保証することができる。

しかし、短所も大きく分けて２つある。

- GC の起動回数を増やすと overhead が大きくなる

  GC によるメモリ回収量を増加させるために、タイマ間隔を短くし GC の起動回数を増やすと、その分 GC の処理のための overhead がかかる。しかし、GC の起動回数を減らすとメモリ回収量が減り、メモリの枯渇が生じる可能性がある。つまり、アプリケーション毎にタイマ間隔の調整が必要となる。(図 3.10)

- タイマの割り込みによる overhead が生じる

  タイマの処理を行うためにも CPU は使用されるので overhead がかかる。

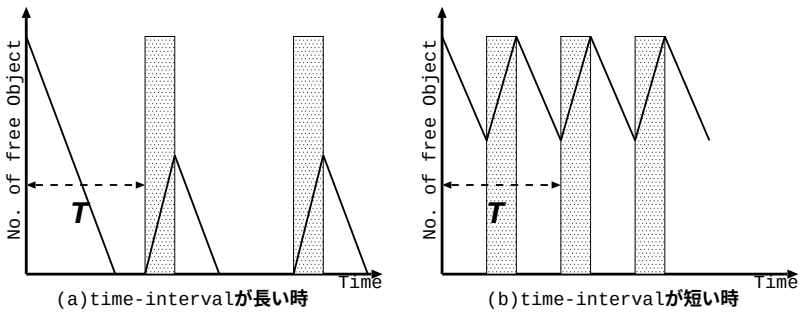


図 3.10: Timed-GC のタイマ間隔

つまり、上記の 2 つのような問題が発生する可能性がある。よって、タイマとインクリメンタルによる overhead を軽減するためには、GC のパラメータをアプリケーション毎に調整する必要がある。

第 4 章

本研究で導入する新方式

この章では、本研究で提案する GC API を説明する。GC API は、次の 3 種類である。

- GC の起動抑制・許可
- GC の実行間隔や優先順位の指定
- メモリ予約

4.1 Garbage Collection API の提案

本論文では、リアルタイム処理に適した機能として、Java のガベージコレクション API(GC API) を提案する。この GC API を用いると、プログラムはリアルタイム制約を満たすためにアプリケーション毎に GC の動作を変更することが可能となる。本来、GC はアプリケーションプログラマに対して透明、つまりメモリ管理を意識させないことが重要であるが、次の理由から GC API が有用であると主張する。

- リアルタイム用の GC アルゴリズムの改良や GC のチューニングだけではリアルタイム制約の充足は難しい。GC の性能はアプリケーション、動作環境、GC のパラメータによっても大きく変化するからである。
- アプリケーションや動作環境ごとに自動的に GC の動作を調整できれば理想的だが、現在の技術では難しい。
- 多くの場合、アプリケーションの開発者はどのように最適化すれば良いかを知っている。また、最適化が必要な部分はコードの一部にすぎないことも多い。それゆえ、

GC API があれば、保守性を保ったまま、そのアプリケーションに特化した GC の動作のチューニングをしやすい。

4.2 GC API の概要

本節では、本研究で提案する GC API を説明する。表 4.1 に GC API の一覧を示す。主に次の 3 つの機能を有する。

- GC の起動抑制・許可

  Write barrier の overhead を無視できないほどリアルタイム制約が非常に厳しい部分には、この API を用いて GC の起動自体を抑制して、デッドラインミスを回避する。

- GC の実行間隔や優先順位の指定

  この API は、各アプリケーションに適した実行間隔を時間単位で指定したり、優先順位を指定する。この考えは、Timed-GC [1] のアイディアを採用している。

- メモリ予約

  GC を持つ実行環境では、メモリの割り当て時に、GC を少し実行させたり、GC に必要な処理 (割り当てたメモリのマーク領域の確保と初期化など) を行うため、一般的に overhead がある。また、ヒープ残量不足で、GC の終了待ちが必要なこともある。つまり、ヒープ残量の保証やメモリ割り当てに必要な時間の予測が難しい。

  このため GC を経由せずにメモリを予め割り当てておきたいことがある。この機能をメモリ予約と呼ぶ。メモリ予約の API は、GC を経由せずに明示的に動的メモリ管理をさせることで、メモリ残量を保証し、メモリ割り当てに必要な時間をより小さく、より予測可能にする。

以下の節では GC API の簡単な使用例を示す。

表 4.1: ガベージコレクション API 一覧

実行間隔の指定	start マイクロ秒毎に period マイクロ秒だけ GC を実行する． public void GC.setInterval (int start, int period) public int[] GC.getInterval ()
優先順位の指定	GC の優先順位を pri とする．範囲は 1〜25 で，1 が最も優先順位が高い． public void GC.setPriority (int pri) public int GC.getPriority ()
起動の抑制・許可	b が false ならば，新たな GC の起動を抑制する．すでに起動中の GC を終了させる効果は無い． public void GC.setInvokable (boolean b) public boolean GC.testAndSetInvokable (boolean b) public boolean GC.isInvokable () public boolean GC.isInvoked ()
メモリ予約	GC の処理対象とならない size KB のメモリを割り当てて，予約番号を返す． public int GC.ReserveMem (int size)
予約の行使	予約番号 id のメモリを使用するか否かを b で指定する． public int GC.ProvideMem (int id, boolean b)
予約の開放	GC.ReserveMem() で予約した予約番号 id のメモリを開放する． public void GC.FreeMem (int id)

4.2.1 GC の起動抑制・許可

アプリケーションの種類によっては実行時に時間的制約の非常に厳しい期間が存在する。この期間内に GC の割り込みが発生するとデッドラインミスが発生する可能性がある。つまり危険性を回避するために GC の起動自体を抑制する。時間的制約の厳しい期間が過ぎれば、GC の起動を許可することで GC の活動が再開される (図 4.1)。

```
// GC が起動中でなければ、アトミックに GC の起動を禁止して
if (GC.testAndSetInvokable(false)) {
    // ここにリアルタイム制約の厳しい処理を書く

    // GC の起動禁止を解除
    GC.setInvokable(true);
}
```

図 4.1: GC の起動抑制・許可の指定

図 4.1 において `GC.testAndSetInvokable(false)` から `GC.setInvokable(true)` の間は GC の起動が禁止されていることから、システムがメモリを滞りなく供給するとは限らない。ゆえに、この期間はユーザがアプリケーションプログラムに対してメモリの供給を保証しなければならない。

図 4.2 は、GC の起動抑制・許可の指定を行わなかった場合と、行った場合を示している。図 4.2 (上) では、GC の起動抑制を行わなかったため、GC の割り込みが発生し、デッドラインミスが引き起こされている。この状態ではリアルタイム制約を充足することはできない。

図 4.2 (下) では、時間的制約の厳しい期間に GC の起動抑制を行う。つまり、図 4.1 で行っている処理を、時間的制約の厳しい期間で行う。つまり、図 4.2 (下) の図を見ても分かる通り GC の割り込みは時間的制約の厳しい期間では起こらないことから、デッドラインミスを回避することが可能となる。

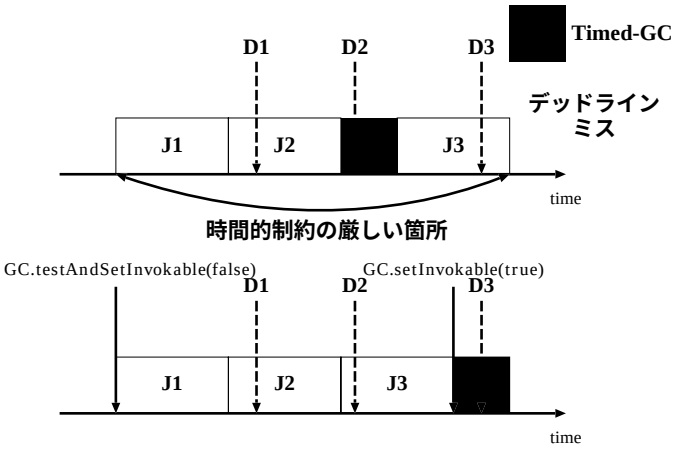


図 4.2: GC の起動抑制・許可

4.2.2 GC の実行間隔や優先順位の指定

Timed-GC は一定間隔の実行間隔で実行される。実行間隔が短ければメモリの回収量は多くなるが、overhead も大きくなる。実行間隔が長くなればメモリ回収量は減るが、overhead は小さくなる。また、実行時間の長さも同様である。 つまり、アプリケーションに対して最適なパラメータを得るためには、GC の実行間隔と実行時間の調整が必要である。

図 4.3 において Time-GC を 100ms 毎に 10ms 実行するように設定している。これは、周期的なアプリケーションであれば、図 4.3 のように GC の周期をアプリケーションの周期に調節することで Timed-GC が効率的にメモリの回収を行ってくれる。なぜなら、周期的なアプリケーションであれば、メモリの使用も周期的に行い、またゴミとなるメモリも周期的に行われるからである。これを図 4.5 に示す。図 (a) は実行間隔と実行時間は不適切に設定されている。つまり、Timed-GC の実行間隔が長すぎることからメモリ枯渇が発生し、また、実行時間が短いことからメモリ回収量が少なくメモリを十分に供給することができない。しかし、図 (b) では、Timed-GC の実行間隔および実行時間が適切に設定されている。この場合、メモリ枯渇が発生する前に十分なメモリを回収することが可能となる。

また、図 4.3 では、Timed-GC の優先度を最大に設定している。これは、Timed-GC の優先度を変化させることで、Timed-GC 以外のスレッドを優先して動かすことが可能となる。我々が実装に用いた JVM である kaffe-1.0.b1-rt[8] では、Java のネイティブスレッドとして、RT-Mach スレッドを用いている。ここで、`GC.setPriority(1)` がセットする優

先度は RT-Mach のスレッドの優先度であり、実装に依存している。(図 4.4)。

```
// 100ms 毎に GC を 10ms 実行するように設定
GC.setInterval(1000000, 100000);

// 優先度を最大に設定
GC.setPriority(1);
```

図 4.3: GC の実行間隔や優先順位の指定

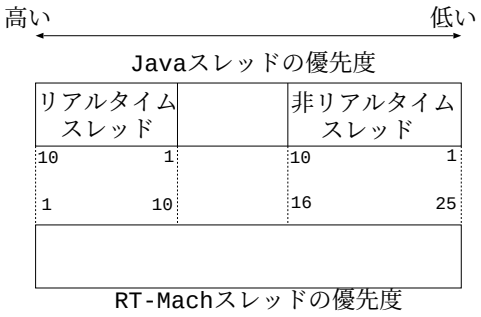


図 4.4: スレッドの優先度

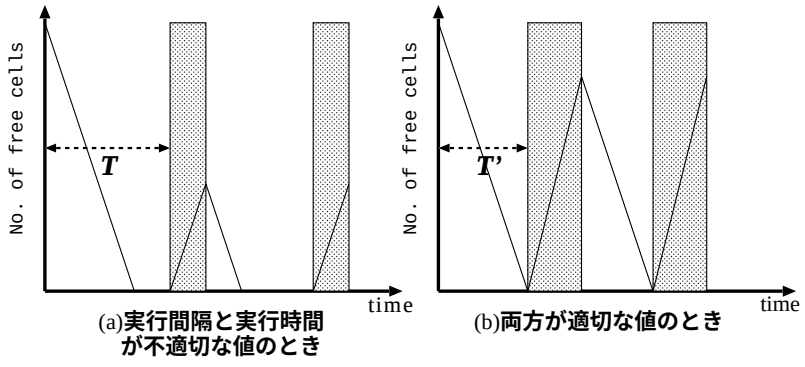


図 4.5: GC の実行間隔と実行時間の変更

4.2.3 メモリ予約

GC に必要な処理にかかるオーバーヘッドや、ヒープ残量不足など GC によってメモリが回収されることを待たなければいけない時がある。この時に時間的制約の厳しい期間となると、アプリケーションはデッドラインをミスしてしまう可能性がある。この場合のため、予めメモリを確保しておき GC を経由することなくメモリを供給することで、メモリ残量を保証し、メモリ割り当てに必要な時間を小さくすることが可能となる。しかし、この領域のメモリの開放を GC に任せてしまうと GC によるオーバーヘッドが依然かかることとなる。ゆえに、この場合はメモリの管理をユーザにまかせ、明示的なメモリ管理を行うようにする (図 4.6、図 4.7)。

```
// 50KB のメモリをヒープ上に予約
int mem_id = GC.ReserveMed(50);

// 予約したメモリからの割り当て開始
GC.ProvideMem(mem_id, true);

// 予約したメモリ上にインスタンスを生成
Foo foo = new Foo();

// 予約したメモリからの割り当て終了
GC.ProvideMem(mem_id, false);

// 予約したメモリの開放
GC.FreeMem(mem_id);
```

図 4.6: メモリ予約の指定

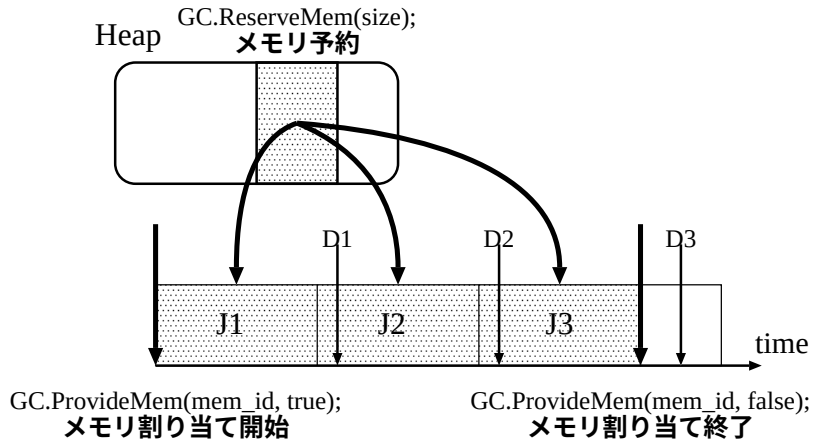


図 4.7: メモリ予約

図 4.6 において、メモリを明示的に 50KB 確保している。このメモリは既に GC の管理下を離れ、ユーザが管理しなければならない。次に、予約 ID を元にメモリを割り当てを開始する。この期間中に生成されたオブジェクトは、予約されたメモリから割り当てられる。また、オブジェクトが使用済となった時、ユーザは明示的にメモリの開放を行わなければならない。

メモリ予約を用いるためにプログラマが保証すべき制約を以下に示す。

- オブジェクトを `GC.FreeMem(mem_id)` 以降で参照しないことを保証する

  GC API のセマンティクスとして、`foo` が参照するオブジェクトの生存期間は `GC.FreeMem(mem_id)` までだからである。

  なお「`GC.FreeMem(mem_id)` 以後は GC の対象となり、参照可能」というセマンティクスにすることもできるが、kaffe-1.0.b1 は conservative GC であることから、これは難しい。

- 外部オブジェクトが予約領域のオブジェクトを参照しないことを保証する

  予約領域を明示的に開放したときに外部からの参照があると、*dangling pointer* が発生する可能性がある。Dangling pointer とは、あるはずのないオブジェクトを指すポインタのことである。この dangling pointer が発生すると、このポインタを用いてあるはずのないオブジェクトに対して書き込みを行うことが可能となる。つまり、セキュリティに問題が発生する可能性がある。

- メモリ予約する際にオブジェクトが必要とする十分の大きさのメモリを確保することを保証する

  メモリ予約では確保するメモリを明示的に指定するが、そのメモリの量で本当にオブジェクトが必要とするメモリの量を満たしているか分からない。つまり、メモリの量が足らなければオブジェクトの生成にミスする可能性がある。

オブジェクトプール

メモリ予約と同様の方式として「オブジェクトプール」という手法がある。この方法では、未使用のオブジェクトを最初に複数生成しておき、必要なときにこのオブジェクトを使用する方法である。この手法は、本研究では採用されていないが有用であると考える。オブジェクトプールの長所を以下に示す。

- リアルタイム制約の緩い期間にオブジェクトを生成しておけば、制約の厳しい期間でオブジェクトを生成しなくても使用することが可能となる。
- オブジェクトはGC の管理下に置かれていることから、使用されなくなったオブジェクトは GC によって回収される。また、メモリ予約と比較して明示的なメモリの開放は必要ない。

また、オブジェクトプールの短所を以下に示す。

- オブジェクトプール用のオブジェクト生成を最初に行うことから、利用するときに生成したオブジェクト管理の保守コストが必要となる。

4.3 議論

この章では、GC API の設計について述べた。本研究で提案した GC API をより良くするには、上記で述べたオブジェクトプール等の他の手法を GC API に取り込んでゆく方法が考えられる。

ところが、我々は kaffe の GC アルゴリズムの大幅な変更を考えてはいない。また、GC API もシンプルなものとしている。なぜなら、GC API を用いて統計的に、アルゴリズムの大幅な変更なしにパラメータ変更だけでどれだけの性能向上が得られるかに興味があるからである。よって、本研究ではそのための環境構築を行っている。

その他の研究、CMM(A Customisable Memory Management Framework for C++)[13]では一般的な解決方法をとっている。すなわち、多種のGCアルゴリズムをユーザに提供することで性能向上を計っている。

しかし、リアルタイムシステム・組み込みシステムでは、GCの規模が大きくなることは非現実的である。つまり、複雑なGCになると処理にかかるoverheadもまた大きくなるからである。

また、その他の研究、RTJEG(The Real-Time for Java Experts Group)[6]などは、GC以外の面からもアプローチを試みているが、我々はGCに着目している。つまり、アプローチ方法も異なっている。

第 5 章

関連研究

この章では、関連研究として RTJEG のリアルタイム仕様と J consortium のリアルタイム仕様および CMM(A Customisable Memory Management Framework for C++) の研究内容について説明する。

5.1 RTJEG のリアルタイム仕様拡張

RTJEG(Real-Time for Java Experts Group) のリアルタイム仕様拡張 (Real Time specification for Java)[6] は、54 ページの仕様書である。RTJEG のリアルタイム拡張では、本研究で提案している GC API と類似した API を持つ。

クラス: GarbageCollector

このクラスは、現在の GC の振る舞いについての情報供給、および GC のパラメータを指定する API を持つ。

- public void `setPreemptionThreshold (int t)`
- public void `setReclamationRate (int rate)`

この API は、まず `setPreemptionThreshold(int t)` によってスケジューリングの敷居値を指定する。次に、`setReclamationRate (int rate)` によって GC によるメモリの回収率を指定する。なぜスケジューリングの敷居値の指定が必要かというと、GC が実行されるスケジューリングの割合によってもメモリの回収率が変化するからである。また、

現在のスケジューリングの敷居値およびメモリの回収率を取得する API も持つ。
(public int `getPreemptionThreshold()`, public int `getReclamationRate()`)

- public int `getBarrierOverhead ()`
- public int `getOverhead ()`
- public int `getPreemptionLatency ()`

この API は、GC によって生じる overhead の量を取得する。つまり、GC によって生じる overhead が大きい場合、GC に割り当てられているスケジューリングを減らしたり、メモリ回収率を減らすことによってリアルタイム制約を充足させる。また GC による overhead が小さければスケジューリングを増やしたり、メモリ回収率を増やすことも可能となる。

クラス: ImmortalMemory, ScopedMemory

このクラスでは、本研究のメモリ予約と類似した機能が定義されている。

- ImmortalMemory

  この ImmortalMemory 領域内に割り当てられたオブジェクトの寿命はアプリケーションの寿命と等しくなる。つまり、この領域に割り当てられたオブジェクトは、アプリケーションが終了するまで消えることは無い。また、この領域内のオブジェクトはアプリケーション終了時まで残ることが保証されていることから GC による影響を受けることはない。

- ScopedMemory

  この ScopedMemory に割り当てられるオブジェクトは、限られた寿命しか持たない一時的に割り当てられたオブジェクトとして扱われる。そして、この ScopedMemory は Java の生成する heap とは別の領域に作られる。

  この領域内のオブジェクトが参照しているオブジェクトの参照数をカウントしておき、他のオブジェクトへの参照が無くなった時に GC が呼び出されて、この領域のメモリを回収する。

この手法では、ImmortalMemory に割り当てられたオブジェクトは回収されないことから、メモリは静的に割り当てられていると考えることができる。また、ScopedMemory

に割り当てられているオブジェクトは最終的にはGCによってメモリを回収する必要がある。つまり、ImmortalMemoryはGCが行われないことからゴミが残り、ScopedMemoryではGCによるoverheadがある。

5.2 J consortiumのリアルタイム仕様拡張

J consortiumのリアルタイム仕様拡張(Draft Internatinal J Consortium Specification)[7]は、128ページの仕様書である。しかし、本研究で提案しているGC APIと類似のAPIを持たない。

この仕様書では、従来のJavaをBaseline Javaと呼び、この仕様書で定義するJavaをCore Javaと呼んで区別している。両者は異なるクラスツリーを形成する。また、実行環境においても両者は異なるヒープを用いると規定されており、Core Javaのオブジェクトが使用するヒープはプログラマによる明示的な制御によって管理される使用となっている。これゆえ、Baseline JavaからCore Javaへのメソッド呼び出し、および逆のメソッド呼び出しは強い制限を課せられている。この点で、RTJEGとはかなり異なる性質の仕様となっている(図5.1)。

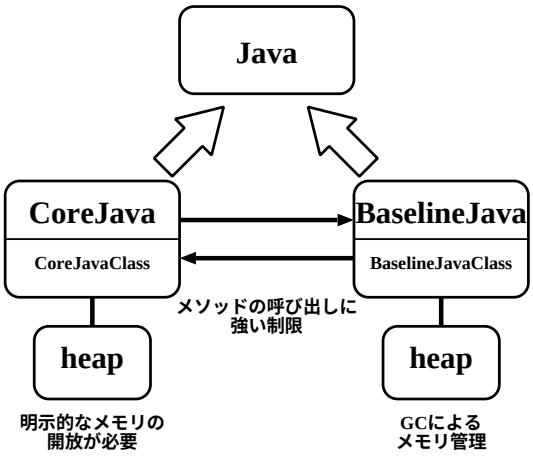


図5.1: J consortiumのリアルタイム仕様

5.3 CMM

CMM(A Customisable Memory Management Framework for C++)[13] は、Giuseppe らによって提案されたメモリ管理のフレームワークである。CMM では、リアルタイム性を考慮したものではないが、GC のカスタマイズが可能となる。

抽象クラス: CmmHeap

CMM では、ヒープの実装のために抽象クラス `CmmHeap` を用いる。

- クラス: `DefaultHeap`

  `DefaultHeap` を用いると、このヒープ内に割り当てられたオブジェクトは mostly-copying generational collection によって回収される。また、このヒープを用いるためには `CmmObject` によって生成されたオブジェクトでなければならない。

- クラス: `TempHeap`

  オブジェクトの大半がプログラムのある期間を過ぎると使用されなくなる時は、`TempHeap` を用いると有効である。なぜなら、ある期間でメモリはほとんど使用されていないことから、このヒープ自体のメモリを開放することが可能だからである。

- クラス: `MarkAndSweep`

  `MarkAndSweep` を用いると、このヒープ内に割り当てられたオブジェクトは conservative mark-sweep GC(2 章 conservative 方式) によって回収される。しかし、メモリの断片化が発生する。

- クラス: `UncollectedHeap`

  `UncollectedHeap` を用いると、ユーザは明示的にメモリの割り当てを行わなければいけない。また、`CmmObject` を継承していないオブジェクトは、このヒープに割り当てられる。また、このヒープに割り当てられたオブジェクトは GC による回収は行われない。

つまり、この 4 つのヒープを用いることで、異なる手法の GC を使用することが可能である。本研究のパラメータを変更するという手法とは異なるが、環境やアプリケーションに対して最も有効な GC を選択することが可能となる。

抽象クラス: CmmObject

CMM では、コレクタによって追跡または回収されなければならないオブジェクトのために `CmmObject` を用いる。回収されるべき全てのオブジェクトは、`CmmObject` からたどることができなければならない。つまり、`CmmObject` 以外で生成されたオブジェクトは `UncollectedHeap` に割り当てられることから、GC によるメモリの回収が行われない。

5.4 議論

この章では関連研究について述べた。ここで、他の研究 (RTJEG、J consortium、CMM) と我々の提案している GC API との比較を行う。

RTJEG との比較

- GC API はライブラリレベルで実現可能

  GC API での変更は GC に特化した部分での変更である。つまり、大幅な変更は必要ない。

- RTJEG は言語レベルでの変更が必要

  RTJEG では GC 以外の部分においても多くの変更が必要となる。つまり、言語レベルでの大幅な変更を余儀なくされる。

J consortium との比較

- GC API は既存の Java に親和しやすい

  GC API では既存の Java に直接変更を加えている。つまり、本来の Java で実行可能なことから、親和性は高いと考えられる。

- J consortium は Java には親和しにくい

  J consortium では、リアルタイムシステム用に異なるクラスを使用する必要がある。これは、リアルタイムシステムには向いているが、本来の Java とはかけ離れており親和しにくいと考えられる。

CMM との比較

- GC API は簡易である

  GC API は複雑な処理を用いておらず簡易に実装される。つまり、GC にかかる負荷はそれほど大きなものとならないと考えられる。ゆえに、リアルタイムシステムや組み込みシステムにおいても利用可能であると考えられる。

- CMM は複雑である

  CMM は複数のヒープを用いて、数種類の GC アルゴリズムを使用することが可能である。しかし、リアルタイムシステムや組み込みシステムにおいては GC が複雑になるのは、処理にかかる overhead が大きくなることから現実的ではない。

第 6 章

GC API の実装

この章では、GC API の実装のベースとして用いている kaffe-1.0.b1 と、リアルタイム化された kaffe、kaffe-1.0.b1-rt についての説明をする。

6.1 kaffe-1.0.b1{-rt} の概要

kaffe-1.0.b1 は C 言語によって実現されており、仮想マシン部分のコードが約 20,000 行、GC 部分が約 2,000 行で構成されている。しかし、kaffe-1.0.b1 から kaffe-1.0.b1-rt を実現する際に、GC 部分には本質的に変更は加えられなかった。以下に kaffe-1.0.b1{-rt} の GC 部分の特徴を示す。

- mark-sweep

  無条件で必要とされるオブジェクトの集合をルートという。mark-sweep は、ルートから到達可能なオブジェクトにマークをつけて、マークされなかったオブジェクトを回収する GC の手法である。(2 章 mark-sweep 方式)

- conservative

  ポインタとデータの区別ができない場合があり、かつ、その場合にポインタとして扱う GC を conservative GC という。ルートの一部であるネイティブメソッドのスタックの扱いを簡単化するため、kaffe-1.0.b1 の GC は conservative となっている。メモリリークが起きること、オブジェクトを別アドレスに移動できないため、copying や compaction ができないという欠点がある。(2 章 conservative 方式)

- two-level allocation

  conservative GC の欠点であるメモリの断片化を軽減するために、動的なメモリの割り当てを２階層にする手法を two-level allocation という。下位レベルではシステムからまとまった大きさで動的にメモリを確保し、上位レベルではそれをブロック毎に固定サイズを決めて、GC にメモリを供給する。(２章 two-level allocation)

また、GC API を実現するために変更を加えた仮想機械で用いられている関数を説明する。

- `gc-incremental.c`

  GC を司る関数。GC の活動を Lock し分割することで incremental 化されている。また、Timed-GC のタイマもここに記述されている。

- `gc-mem.c`

  オブジェクトへメモリを供給する関数。ユーザプログラムからの要求があれば、ここでメモリが割り当てられる。また、メモリ領域はあるがヒープ領域が足らなくなった場合、ヒープ領域は拡張される。

- `thread.c`

  スレッド関係を司る関数。デーモンスレッドの生成 (GC スレッド、Finalizer スレッド、Timer スレッド) や、ユーザスレッドの生成等を行う。

- `internal.c`

  kaffe 固有の関数を RT-Mach 化するための変更等を行っている。RT スレッドの生成、優先度の変更や snapshot を行うための関数が記述されている。

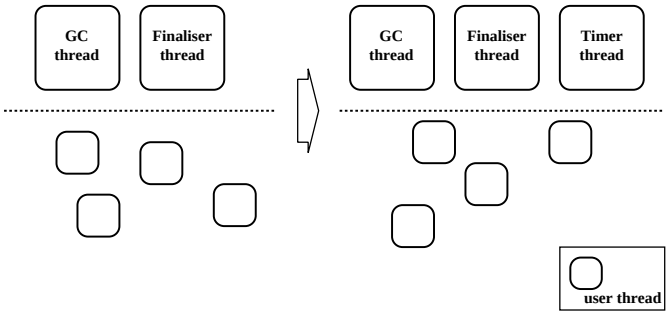
6.2 GC API の実現の概要

kaffe-1.0.b1{-rt}

表 4.1 の GC API を実現するために、kaffe-1.0.b1{-rt} の GC に以下の修正を行った。

- 湯浅のスナップショットアルゴリズム [2][11] を用いて並行 GC とした。

- write-barrier はメモリ保護機構と RT-Mach の例外処理機構を用いた OS 支援方式で簡易に実現した。
- ヒープ残量だけでなく、一定時間毎に GC を動作させるためにタイマスレッドを追加した (図 6.1)。



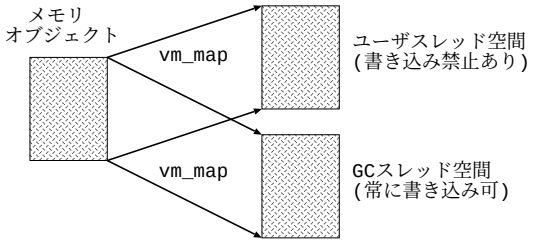


図 6.2: **vm_map** によるメモリ空間の分割

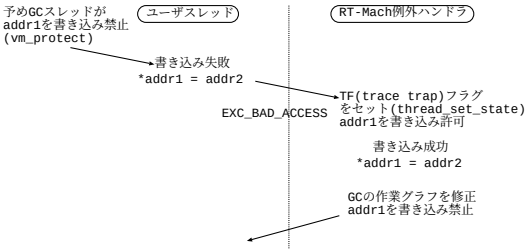


図 6.3: write-barrier の概要

GC API

GC の起動抑制・許可

GC の起動抑制・許可の変更を行う API を実現するために以下の修正を行った。

- GC の起動抑制・許可の指定を実現するため、GC スレッドを停止することで起動抑制・許可を行うように変更を加えた。

図 6.5 でユーザプログラムからの起動抑制の命令があると Native method 内で Timer thread から現在の GC の状態を取得する。もし、GC が起動されていなければ GC の起動抑制を行う。ここで、RT-Mach 内フラグ変数を用意し、フラグ変更参照を排他的に行う。つまり、他のスレッドからの GC の起動要求や明示的な GC の起動 (例えば `System.gc();`) を無視する。ゆえに、GC 起動のための処理を行うことができないので、GC は起動抑制される。

また、ユーザプログラムから起動許可の命令があれば、同様にして Timer thread に GC 起動許可の処理を行う。

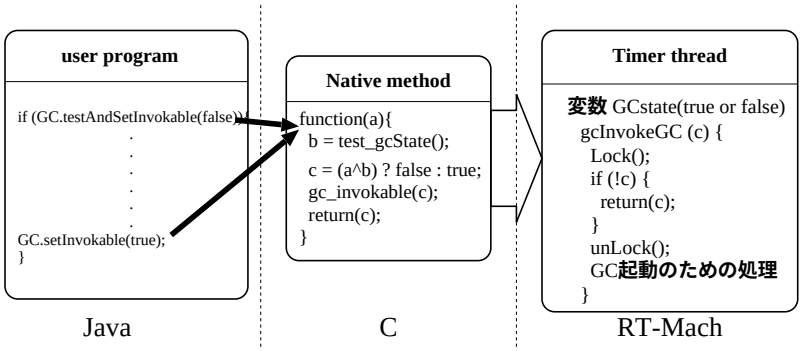


図 6.4: GC の起動抑制・許可指定の概要

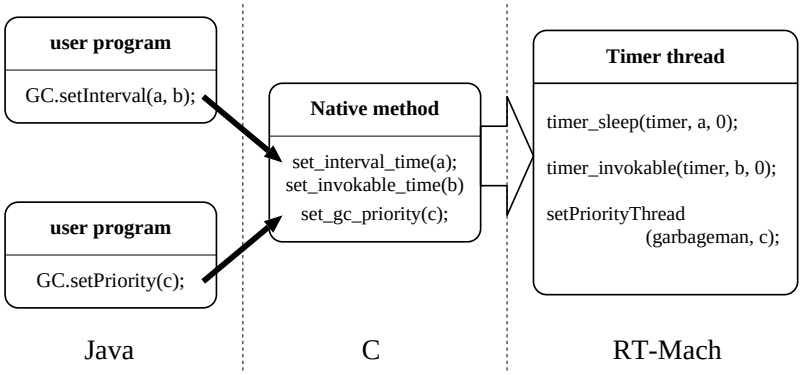
GC の実行時間や優先順位の指定

GC の実行時間や優先順位の指定の変更を行う API を実現するために以下の修正を行った。

- GC の実行時間の指定を行うためタイマスレッドに変更を加えた。
- GC の優先順位の指定を行うため GC のスレッド ID を取得し、RT-Mach スレッド用の優先順位へと変更した。

図 6.5 でユーザプログラムから実行間隔の変更命令があれば Native thread 内の関数を呼び出して Timer thread の実行間隔を変更する。GC は、`timer_sleep(timer, a, 0);` の間は起動されず、`timer_invokable(timer, b, 0);` の間は起動される。つまり、この間隔を調整することで Timer thread の実行間隔は変更される。

また、ユーザプログラムから優先度の変更命令があれば、`setPriorityThread(garbageman, c);` によって GC の優先度の変更を行う。



メモリ予約

メモリ予約のための実装はまだ行っていないが、実装に要求されることを以下に示す。

- 予約されたメモリの確保

  ヒープ上に GC の領域とは別にメモリ予約用のメモリを確保する。しかし、GC 用の malloc、*gcMalloc* を用いてメモリを確保してしまうと、予約メモリは GC の管理下に置かれてしまう。ゆえに、gcMalloc の他にメモリを明示的に管理するための malloc, *gcMalloc2* を用意する必要がある。つまり、図 6.6 の gcMalloc の手前から分岐して gcMalloc2 を実装することで実現可能となる。これは、`gc_heap_malloc` の段階ではメモリは GC の管理下に置かれていないからである。

- 予約メモリの割り当て

  メモリ予約時にメモリの予約 ID が返り値として戻る。この予約 ID を元にメモリを割り当てる。Java では、`new` することで新たに生成したオブジェクトにメモリを供給するが、ここで割り当てられるメモリは GC の管理下のメモリである。つまり、予約メモリを割り当てるときの `new` は、通常の `new` であってはならない。ゆえに、予約メモリを割り当てるときは `new` のメモリ供給元を予約メモリに変更する必要がある。

- 予約メモリの開放

  予約メモリを開放する場合、予約 ID を元に予約領域をすべて開放する。よって、予約 ID から得られるヒープ上の予約メモリへのアドレスを C 言語の `free` によって開放できるように `GC.FreeMem(mem_id)` を実装することで実現可能となる。

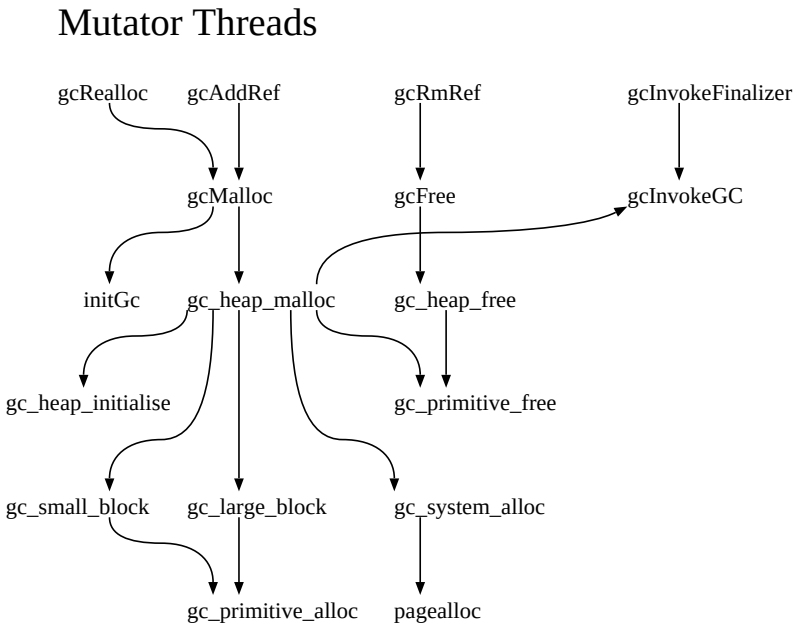


図 6.6: GC におけるメモリ関係の関数のつながり

6.3 プログラム例

ここでは、GC API を用いていないが、並行 GC であることからデッドラインミスが少なくなるプログラム例を示す (図 6.8)。

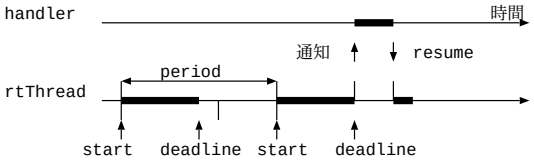


図 6.7: リアルタイムスレッドとデッドラインハンドラ

このプログラム中のリアルタイムスレッド (`rtThread`) は、周期的に動作し、指定したデッドラインを満たせないと、デッドラインハンドラのメソッド `Handler` を起動する (図 6.7)。このプログラムを kaffe-1.0.b1-rt で動作させると、35ms では GC が終了しないため、10 回の繰り返しで 10 回のデッドラインミスが起きた。我々が実現した並行 GC では、`System.gc()` はブロックしないので、デッドラインミスは 1 回のみであった。この 1 回はクラス `System` のロードが原因である。

```
import kaffe.rt.*;
class myProgram {
    public static void main(String args[]) {
        // デッドラインハンドラ(スレッドの一種)を作成して起動
        MyHandler handler = new MyHandler();
        handler.setPriority(10);
        handler.start();
        // タイミング属性の設定
        Time period   = new Time(1, 0);        // 1秒
        Time deadline = new Time(0, 35000000); // 35ミリ秒
        Time start    = new Time(0, 0);        // 0秒
        MyRtThread rtThread = new MyRtThread();
        // スレッドの属性を設定
        rtThread.setAttr(1,         // 優先度
                         start,     // 開始時間
                         period,    // 周期時間
                         deadline); // デッドライン
        // ハンドラオブジェクトとメソッド名の指定
        rtThread.setHandler(handler, "Handler");
        rtThread.start();
    }
}
class MyHandler extends RtHandler {
    public void Handler(RtThread Th) {
        // デッドラインミスしたスレッドを再開
        Th.resume();
    }
}
class MyRtThread extends RtThread {
    public void run() {
        System.gc(); // 周期的にGCを起動
    }
}
```

図 6.8: GC の実行間隔や優先順位の指定

6.4 議論

6.4.1 GC API の実装で得られるもの・失われるもの

使用環境に適応させるため GC API を実装した。GC API を付け加えることで得られるものがあったが、逆に失われるものもある。これを以下に示す。

GC API によって得られるもの

- GC のパラメータ調節

  GC の性能を使用するアプリケーション、動作環境により調節可能となり、GC の性能向上につながる。

- リアルタイム制約の充足の向上

  GC API を用いることで、リアルタイム制約の厳しい期間であれば GC の活動を抑制するなどしてリアルタイム制約を充足しやすくなる。

GC API によって失われるもの

GC API を実装するために、ネイティブメソッドを用いて JVM に変更を加えた。また、kaffe-1.0.b1 の GC を並行 GC に変更した。しかし、この変更をしたために失ったものを以下に示す。

- セキュリティ

  `GC.FreeMem` 後に予約したメモリ上のオブジェクトへの参照を使用できる。つまり、あるはずのないオブジェクトへの書き込みが可能となり、セキュリティに問題が発生する。

- ポータビリティ

  GC API を付け加えるためにネイティブメソッドを使用したことから移植性を失うこととなる。

6.4.2 GC のコードは保守性が低い

GC API の実現の作業では、予想以上にデバックの作業が困難であり、多くの時間を占めている。このため我々は「GC のコードの保守性は低い」と強く感じている。この原因は (我々のスキルの低さもあるが) 以下にあると分析する。

- アルゴリズムの内容に比べて、コード量が多い。つまり、非機能的用件を実現するコード部が多い。言い換えると GC はチューニングの塊である。
- コード部分同士の結合度が大きく、コード変更の影響が広範囲に渡りやすい。
- GC の性質上、バグの原因を特定しにくい。GC が誤って使用中のオブジェクトを回収すると dangling pointer の原因となる。これは GC のバグが原因だが、ユーザスレッド中でのメモリ不正アクセスとして現れる。また、メモリリークは発見そのものが難しい。
- 並行 GC なので、並行プログラムとしてのデバックの難しさもある。

このため、GC の効率を犠牲にせず、保守性を高く保つ技術、例えば GC 用のアスペクト指向型言語が必要だと考えている。

6.4.3 GC API は実装に強く依存する

一般的に API は実現に対して独立であるが、GC API は GC の実装に強く依存するし、依存すべきであると我々は考えている。GC の実装ごとに、調整できるパラメータや、GC の動作を変更する操作の集合が異なるので、無理な API の統一は有効なチューニングを妨げると考えるからである。GC API をうまく依存/非依存で分離して、見通しは良くする必要があるが、本論文ではこの問題を無視している。

第 7 章

おわりに

本章では本研究のまとめを行い、この研究で得られた結論・今後の課題について説明し、最後に将来の展望について述べる。

7.1 まとめ

本研究を行うにあたって GC の基本的な概念からはじまり、GC の基本的なアルゴリズム (例えば reference counting、mark-sweep、copying)・リアルタイム環境で一般的に用いられるアルゴリズム (例えば incremental、generational) の解説を行った。

その後、本研究で提案する GC API の設計について解説することでリアルタイムシステムにより求められる GC の性能を述べ、GC がリアルタイム制約を満たす手段として 3 つの GC API(GC の起動抑制・許可、GC の実行間隔や優先順位の指定、メモリ予約) の提案を行った。また、部分的ではあるが GC API の実装を行った。

また、本研究に関連する研究 (RTJEG、J consortium、CMM) の研究内容と、本研究で提案する GC API との比較を行い、GC API の有用性を述べた。

本論文では、リアルタイム処理に適した機能として、Java のガベージコレクション API(GC API) が有用であると主張した。また、kaffe-1.0.b1-rt に基づく実装方法を示し、基本的な実現可能性を確認した。

7.2 結論

本研究では、GC API の部分的な実装を行ったがデバッグやチューニング不足から、GC API を用いた定量的なデータは取得できなかった。しかし、本研究を通じて GC の保守

性の低さ、GC の難しさが理解できる。また、リアルタイム性を加味することによって、なおいっそう保守が難しくなることが分かる。加えて、GC API を用いることで生じる問題も発生する。つまり、６章で述べた通りセキュリティやポータビリティについても考慮する必要がある。

本研究はまだまだ中途段階にあり、本研究の有用性は確認できていない。GC API の実装を完成させた上で GC API によるリアルタイム性の改善の実例を示す必要がある。

7.3 今後の課題

- メモリ予約の実装

  メモリ予約の設計は終了しており、実装は可能である。

- GC API によるリアルタイム性改善の実例

  リアルタイム性が改善された実例を挙げるために、GC 自身のデバックを行い GC API を稼働させる必要がある。また、GC API の有効性を証明するため、実験用アプリケーションが必要である。

- セキュリティの問題

  本論文ではセキュリティ面の問題よりも先に、１章で述べた以下の要求が充足すべきであると考えている。

  (1) リアルタイム性を満たすこと

  (2) メモリを滞りなく供給すること

  (3) コードが理解しやすく、保守しやすいこと

  このため、本論文ではセキュリティの問題は無視している。

- ポータビリティの問題

  ６章で述べたポータビリティ面での問題は、ネイティブメソッドを用いることで発生している。本研究では、実現のしやすさに着目していることから現時点では移植性が損なわれている。そこで、GC API を標準化するなど移植性を高める必要がある。

7.4 将来の展望

GC API の将来の展望として、GC のパラメータ自動調整がある。アプリケーションや動作環境ごとに自動的に GC の動作を調整できれば理想的であるが、現在の技術では難しい。しかし、GC API の実験を通じてアプリケーションや動作環境ごとのパラメータ調整が有効であると確認されれば、得られたデータから GC のパラメータ自動調整の足掛かりとなると期待できる。

もし、GC のパラメータ自動調整が可能となればユーザはメモリ管理を気にすることなくリアルタイムアプリケーションの使用・作成が容易に可能になると期待できる。

謝辞

本研究を行うに当たり、有益な御指導、助言および援助を頂きました権藤克彦先生に深く感謝の意を表します。また、本研究についてさまざまな議論をして頂いた片山卓也先生をはじめとするソフトウェア基礎講座のメンバーの皆様方に深く感謝致します。

参考文献

[1] Ito, T.and Sasaki, H., Timed-GC for a real-time Lisp system, Proc. ACM SIGPLAN Workshop on Languages, Compilers and Tools in Real-Time Systems, 1997

[2] Richard Jones and Rafael Lins, Garbage Collection, John Wiley & Sons, ISBN 0-471-94148-4, 1996

[3] 岩井 輝男, 中西 正和, Snapshot 型並列 GC におけるルート挿入時間の削減, 情報処理学会論文誌プログラミング, Vol40, SIG4(PRO3)

[4] 組み込みシステム用基板ソフトウェア SMAF の研究開発ホームページ http://www.mkg.sfc.keio.ac.jp/smaf/

[5] James Gosling, Bill Joy, and Guy Steele, 村上 雅章 訳, The Java 言語仕様. アジソン・ウェスレイ・パブリッシャーズ・ジャパン, 1997

[6] The Real-Time for Java Experts Group, Real Time Specification for Java ver 0.8.1, 1999, http://www.rtj.org/rtj.pdf

[7] J Consortium, Real-Time Core Extension for the Java Platform (Draft 1.0.2), http://www.j-consortium.org/spec.PDF

[8] A. Miyoshi, T. Kitayama, H. Tokuda: Implementation and Evaluation of Real-Time Java Threads, 18th Real-Time Systems Symposium, pp. 166-175, 1997

[9] Transvirtual Technologies, Inc., Kaffe OpenVM, http://www.transvirtual.com/

[10] H. Tokuda, T. Nakajima, and P. Rao. Real-Time Mach: Towards a Predictable Real-Time System. Proc. USENIX 1st Mach Workshop, Octorber 1990.

[11] Taichi Yuasa, Real-time garbage collection on general-purpose machines, Journal of Software and Systems, 11(3), pp.181-198, 1990.

[12] 風間 一洋, Java の最新技術動向, Unix Magazine 1 月号, pp.36-49, 2000.

[13] Giuseppe Attardi, Tito Flagella and Pietro Iglio, A Customisable Memory Management Framework for C++, Software-Practice and Experience, VOL. 28(11), 1143-1183(SEPTEMBER 1998)