修 士 論 文

組み込み機器のための
メモリ管理における断片化の改善について

北陸先端科学技術大学院大学
情報科学研究科情報システム学専攻

高橋 毅

2002 年 3 月

修 士 論 文

組み込み機器のための
メモリ管理における断片化の改善について

指導教官　権藤克彦 助教授

審査委員主査　権藤克彦 助教授
審査委員　片山卓也 教授
審査委員　日比野靖 教授

北陸先端科学技術大学院大学
情報科学研究科情報システム学専攻

910060 高橋 毅

提出年月: 2002 年 2 月 15 日

目 次

概 要

本論文では、組み込み機器に適したメモリ管理方法の提案および、プログラムのテスト環境の構築を目指す。近年の半導体技術の進歩に伴い、組み込み機器の大規模化、複雑化が急激に進み、組み込み機器の開発に関しても、Java を使いたいという要望がある。なぜならば、Java は言語仕様にポインタがないことや、ガベージコレクタなどよい性質をもつからである。その要望に答えるべく、小型の Java VM が登場しつつある。本研究で対象とする小型 Java VM である KVM は、バイトコードがコンパクトであること、ネットワーク経由によるプログラムの変更が可能であるなどの良い性質をもつ。このことから、携帯電話や PDA などの組み込み機器に利用されはじめている。

KVM が対象とする小型デバイスは、数年前の標準的なコンピュータに匹敵するほどの性能をもつ。しかし、プログラマはデスクトップコンピュータで動作する Java VM よりも、メモリ管理を慎重に行わなければならない。なぜなら、デスクトップコンピュータで動作するような大量にメモリを消費するプログラムを、限られた資源しか持たない小型デバイスで動作させることが難しいからである。また、KVM のメモリ管理はコンパクションをしないため、断片化を引き起こす可能性が高いことや、非インクリメンタル GC であるため GC が開始されると計算が中断するという欠点をもつ。特に前者の問題は、長時間の動作に耐えなければならないような組み込み機器では致命的な問題となる。

しかし、この問題を改善するため、メモリ管理方法に改良を施したとしても、使用するアプリケーションや確保できるヒープ容量の大きさによって、メモリの使用効率は大きく変化する。つまり、単一のアルゴリズムの改良だけでは大きな性能向上は難しい。

よって、本研究では組み込み機器に適したメモリ管理方法を提案し、テスト実行することで適切な割り当て方法とパラメータを調整できるようなテスト環境の構築を目指す。このテスト環境を用いるとプログラマは、最もメモリ使用効率の良い割り当て方法を導くことが可能となる。また、テスト環境では、メモリ内部の状態 (空き領域なのか、使用されている領域なのか) を視覚的に見れるため、理論的にも直感的にもメモリ割り当てのアルゴリズムを改良し、実験することができる。

本論文では、既存の割り当て方法や、組み込みシステムに適した割り当て方法に対する要求や設計、および実験で得られたことについて説明する。

第1章　はじめに

1.1　背景

近年の半導体技術の進歩に伴い、組み込み機器の大規模化、複雑化が急激に進んできた。しかし、パソコンやワークステーションのようなデスクトップ分野と組み込み機器では置かれる立場が異なる。デスクトップ分野で大規模なアプリケーションを動作させる場合、その動作が満足されるものでなければアプリケーションではなく、デスクトップ分野の性能が低いと判断されることが多い。これに対して組み込み機器では、アプリケーションが大規模すぎると判断されることが多い。また、デスクトップ分野には高い汎用性が要求されるが、組み込み機器では必ずしもそうではない。

しかし、現在の携帯電話やPDAなどの携帯端末は、10年前の標準的なコンピュータに匹敵する性能をもつようになり、性能の異なる携帯電話やPDA上で同じアプリケーションを動作させることが要求されはじめている。つまり、組み込み機器にも汎用性が求められるようになってきた。ところが、組み込みの開発環境は貧弱で、開発言語もアセンブラやC言語が中心である。製品に高い付加価値が求められ、多機能化が進んでいる現在、プログラムがより大規模で複雑になり、従来のプログラム開発では開発が困難となりつつある。このような理由で組み込み分野の開発環境に、技術革新が求められている。その要求の答えとして有望視されているのがJavaである。

近年、プログラミング言語Javaは広く利用されており、Javaの以下のような性質から、組み込み機器まで応用範囲が広がりつつある。

- 高い移植性
  C言語やPascalのように言語仕様に実装依存がなく、バイトコードはどのような環境上でも共通の形式であるため、他の言語仕様に比べて移植性が高い

- ポインタの機能制限
  一般的に、C言語でプログラムを組んだ場合、バグの多くがメモリリークやダングリング参照などポインタによるものでだといわれている。Javaの場合、ポインタの機能を大幅に制限して、このようなバグをの問題をかなり改善している。

- ガベージコレクタがある
  C言語では、プログラマにメモリ管理の責任がある。しかし、Javaではガベージコレクタが不要になったオブジェクトを自動的に回収するため、プログラマはメモリ管理をしなくても済む。

- ネットワーク経由でプログラムの変更ができる
  従来の組み込み機器では、プログラムが ROM に焼きこまれているので、少しプログラムを変更する場合でも ROM を交換する必要があった。しかし、Java の場合、ネットワーク経由でプログラムを配信する機能をもつので、プログラムの変更をネットワーク経由で実現することができる。

- コンパクトなバイトコード

しかし、従来の Java VM では容量が大き過ぎ、組み込みへの応用は難しい。この問題に対処し、組み込み機器への Java の応用を可能にするのが小型 Java VM の KVM である。KVM はポケットベル、携帯電話などの小型でリソースに制約のあるデバイス用に最初から設計された Java の実行環境で、160KB のメモリ領域で効果的に動作する。

KVM(CLDC 1.0) の提供しているメモリ管理部分は、GC が単純な mark-sweep ガーベジコレクタで、一般に比較的メモリが小さい場合に有効である。GC は開発者にとって非常に有用な機能であるが、KVM の GC はコンパクションをしないため、メモリの断片化がヒープ領域を圧迫する可能性をもつことや、非インクリメンタル GC であるため、GC が開始されると計算が中断してしまうという問題をもつ。特にコンパクションをしないという問題は、長時間の動作に耐えられない可能性をもつため、改善が望まれる。

また、メモリの制約によりデスクトップ環境で動作するような大量にメモリを消費するプログラムを動作させることは難しい。従って、デスクトップコンピュータで動作する Java VM よりもメモリ管理を慎重に行わなければならない。また、メモリ管理方法に改良を施したとしても、使用するアプリケーションや確保できるヒープ領域の大きさによって、メモリの使用効率は大きく変化する。つまり、単一のアルゴリズムの改良だけでは大きな性能向上は難しく、アプリケーションごとに割り当て方法を変更することが望ましい。

断片化の問題を最大限に解決するためには、断片化の発生する可能性を軽減させれば良い。なぜなら、メモリの再利用性の向上が期待できるからである。断片化の発生を軽減する方法は 2 通りある。1 つは GC にコンパクションを導入することで、もう 1 つは割り当て方法の改善することである。

メモリ割り当てに関する研究は多く、断片化を改善する方法もいくつか提案されている [2]。ところが、同じ環境下、特に組み込み機器で割り当て方法を比較した研究は見あたらない。また、一般的に割り当て方法を理論的に性能を導くことは難しいため、組み込み機器に適した割り当て方法を提案するには、多くの実験をすることが重要となる。

1.2 目的

本研究ではコンパクションをせずにメモリの断片化の可能性を軽減する割り当て方法の設計 · 評価および、実験をするためのテスト環境の構築を目的としている。このテスト環境を用いると、プログラマはメモリの使用効率の最も優れた割り当て方法を実験的に知る

ことができる。本来、割り当て方法は単一であり、アプリケーション毎に割り当て方法を変更することはない。しかし、以下の理由から、アプリケーション毎に割り当て方法を変更することが有効であると主張する。

1. **組み込み機器の環境によって、割り当てアルゴリズムの性能が違う**
   組み込み機器は、ゲームから航空制御まで幅広い。本研究では、対象とするデバイスを Palm としたため、幅は狭くなった。とはいえ、10KB 代〜512KB というようにデバイスによって確保できるヒープ領域に大きな差がある。Two-level allocotion などの割り当て方法は、内部断片化を発生させることにより、利用効率の向上を狙っているが、最大で 50 %の領域が無駄になる。よって、10KB のヒープ領域しか確保できないのであれば、外部断片化は発生するが、内部断片化の発生しない Fit 法の方が適している場合も考えられる。
2. **割り当てアルゴリズムによって、アプリケーションの動作がかわる**
   アプリケーション毎に、割り当てるオブジェクトの大きさが違う。また、割り当てるオブジェクトの大きさ、数量が同じでも、割り当てる順番が異なると、アプリケーションの動作が変わるため、割り当て方法に一般性を持たせることは難しい。もちろん、同じアプリケーションでも入力毎に特徴が変わる。しかし、例えば、24byte の領域を無数に要求するようなゲームアプリケーションに、Two-level allocation や Binary buddy system を適用すると、一般的な 2^n 単位の大きさで割り当てる場合、割り当て毎に 8byte の領域を内部断片化する。したがって、この場合は 24byte の領域をあらかじめ確保することで、記憶領域の使用効率の向上が見込める。
3. **組み込み機器では、単一のアプリケーションしか扱わない場合もある**
   PDA や携帯電話上では複数のアプリケーションが動作するであろうが、組み込み機器では単一の目的のしか持たないものもある。このような組み込み機器上で動作するアプリケーションは、専用の割り当て方法を適用する方が良い場合が多い。なぜなら、より小さなメモリでアプリケーションを動作させることで製造コストを下げられるからである。

断片化を改善する方法は 2 つある。一つはコンパクションを導入することで、もう一つは断片化を軽減する割り当て方法を KVM に導入することである。本来、GC 処理でコンパクションをおこなうことが断片化を改善する最良の方法である。しかし、GC は非常に有効な技術であり必要であるが、コンパクションの導入は、領域をコピーしてずらさなければならない。しかし、KVM で領域を移動させるのは難しい。なぜなら、KVM は C 言語で実装されているためオブジェクトとポインタの区別がつかないからである。コンパクションを導入できたとしてもコンパクションの処理で、ユーザプログラムの停止時間がさらに増加する。コンパクションの処理を分散すれば、停止時間は元に戻るが、ユーザプログラムの実行が全体的に遅くなる。また、GC の変更は割り当て方法の変更よりも技術的に複雑である。よって、割り当て方法の変更と比べるとバグの発生する可能性が高くなる。従って、組み込みという環境には必ずしもコンパクションが有効とは限らない。逆

に、断片化を軽減する割り当て方法を導入する場合、Linux に導入されたスラブアロケータのように、アロケータの呼び出し回数を減らしたり、頻繁に使われるオブジェクトをまとめてキャッシュするというような単純なアイデアで良い性能を出す場合もある。また、断片化を減少させることは、コンパクションを導入したとしても、コンパクションの回数を減らす可能性をもつ。よって、本研究ではテスト環境を構築し、そのテスト環境を用いて様々な割り当て方法を実験し、各環境に適した割り当て方法を導く。また、デスクトップ分野と比べ、組み込み機器では、長時間の実行に耐えなければならない場合が多いため、長時間の動作に耐えることのできるアロケータの構築を目指す。

また、組み込み機器のようなメモリの小さな環境に、様々な割り当て方法を導入し、実験・調査を行うことは今後の組み込み機器の割り当て方法の性能向上に貢献できると考えられる。このことから、本研究ではコンパクションせずにメモリの断片化の可能性を軽減し、小さなメモリ上でも軽快に動作するアロケータの実現を試みる。

1.3 実現の概要

本研究において KVM という小型 Java 言語システムを使用した。しかし、メモリ割り当ては基本的な Fit 法 (空き領域から、要求された領域を切り取る) であるので断片化の増加を防ぐことは難しい。また、メモリ管理方法に改良を施したとしても、使用するアプリケーションや確保できるヒープ領域の大きさによって、メモリの使用効率は大きく変化する。よって、テスト環境を構築し実験によって、各アプリケーションに適した割り当て方法、パラメータを導出するテスト環境の構築をおこなう。

断片化を減少させるために本研究で注目している事項は以下であり、このことを踏まえて様々な割り当て方法についてチューニングを施す。

- **再利用性を高めるために、内部断片化の可能性を承知する**
  内部断片化を発生させると、その領域が細分化することを防ぐことができる。よって、比較的大きな領域を絶えず確保することができるからである。

- **再利用性を損なわない範囲で、内部断片化の減少を目指す**
  内部断片化を発生させると、再利用性が高まる可能性をもつ。しかし、例えば、2^n の大きさごとに領域を確保すると最大で 50 %の領域が無駄になる。よって、再利用性を損なわない範囲で、内部断片化の減少を目指すことで利用効率を高めることができる。

- **要求の頻度が高い大きさのオブジェクトとその他のオブジェクトで割り当て方法を変える**
  要求の頻度の高いオブジェクトに関しては、その大きさちょうどの領域をあらかじめ確保することで、内部断片化を 0 にすることができる。

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：一般的なメモリ割り当て方法 (Fit 法、Two-level allocation、Buddy system) を説明し、実際に Linux2.2 で実装されているスラブアロケーションについて説明する。

3 章：本研究で提案するメモリ割り当て方法 (Triple buddy system、Block buddy system、Separate first fit 法) について説明する。

4 章：本研究で構築したテスト環境の解説を行う。このテスト環境は、乱数を用いて各割り当て方法について実験する環境である。視覚的にメモリ内部の状態を表示するので、理論的なチューニングの他に感覚的なチューニングを施すことができる。

5 章：本研究で提案する割り当て方法を含め、様々な割り当て方法についてパラメータや全体の記憶領域の大きさを変更し、実験を行った。この実験結果を示すとともに考察をおこなう。

6 章：関連研究として、処理系とメモリ管理部分 (GC) を独立させ、アプリケーションごとに異なるメモリ管理方法を実装することを目的にした研究と、断片化を減少させることを目的とした研究についての説明し、本研究との違いを述べる。

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、達成できなかったことについて述べる。また、これからの展望について説明する。今回の研究では、特に内部断片化は発生する割り当て方法が安定した結果を残した。

付録：直接的に本研究には関係ないが、メモリ管理という意味で密接に関係しているガベージコレクションについて簡単に説明する。

別冊：本研究で対象としている小型 Java VM である、KVM についてメモリ管理部分を重点に説明する。ライセンスが存在するため、別冊で説明する。

第2章　既存のメモリ割り当て方法

この章では、基本的なメモリ割り当て方法に関する用語とアルゴリズム (Fit 法、Two-level allocation、Buddy system) の説明、実際に Linux で実装されているスラブアロケーションについての説明をする。まず、基本用語を説明する。

外部断片化

ヒープ領域は、最初ひとつづきの空き領域となっているが、様々な大きさの記憶領域を切り出して割り当てられる。割り当て·解放を繰り返すと、ヒープ領域中の空き領域は比較的小さな領域に分断されて散らばってしまう。この状態のもとで、分断された領域よりも少しでも大きな領域が要求されると、それが空き領域の総和よりも小さな領域であったとしても割り当てることはできない。この現象を外部断片化 (External fragmentation) という。

内部断片化

外部断片化を避けるひとつの方法として、小さな空き領域が残らないように、要求された大きさ以上の固定の領域に割り当てる方法がある。固定の領域は、例えば 16byte で、9byte から 16byte までのオブジェクトを 1 つ割り当てると決める。大きさを固定にすることで、オブジェクトが分割されずに保存されるため、外部断片化は避けられる。しかし、割り当てられた個々の記憶領域に使用されない無駄な領域が分散することになり、やはり記憶効率を低下させることになる。この現象を内部断片化 (Internal fragmentation) という。

コンパクション

割り当て·解放という動作を続けると、ヒープ領域中の空き領域が比較的小さな領域に分断されて散らばってしまう。この問題を解決するために、ヒープ内でデータを移動して、大きな空き領域が 1 つだけあるような状態にすることをコンパクションという。基本的な方法は 2 つで、1 つは、同じ大きさの二つの領域を用意し、通常は片方だけ使用する。片方の領域が一杯になると使用中のオブジェクトだけをもう片方の領域へコピーする方法である。もう 1 つは、領域を分割せずに使い、使用中の領域だけを一方の端へまとめる方法である。

2.1 Fit 法

大きさ n の領域が必要な場合、n 以上の大きさからなる空き領域が存在すれば、その空き領域にオブジェクトを割り当て、残りの領域を空きリストに加えればよい。しかし、大きさが n 以上の空き領域が複数存在する場合、どの空き領域にオブジェクトを割り当てるかが問題となる。この問題に対し、一般的には次の 4 種類の割り当て方法が存在する。

First fit 法

First fit 法は、大きさ n の領域が必要な場合、空きリストの先頭からを順にたどり、大きさが n 以上の領域が存在すれば、その空き領域にオブジェクトを割り当てる。外部断片化を避けるために、残りの領域が一定の大きさ以下となる場合には、空きリストに加えずにそのまま全体を割り当てることもある。

しかし、First fit 法は空きリストの先頭部分の領域を細分化し、大きな空き領域が残りにくいという傾向がある。よって、比較的大きな領域が要求された場合、条件に合う領域が見つかるまでの時間が増加する。この欠点を克服するために、Next fit 法が提案されている。

Best fit 法

Best fit 法は、大きさ n の領域が必要な場合、空きリストを順に全てたどり、大きさが n 以上で、かつ最も n に近い大きさの空き領域にオブジェクトを割り当てる。大きな領域を、後での使い勝手を考えて保存しておくことは、良い方針であると考えるのが自然である。歴史的にも Best fit 法が長年にわたって用いられた [5]。

しかし、Best fit 法にもいくつかの欠点がある。1 つは、空きリストを全てたどるので First fit 法と比べて一般的な解釈では 2 倍程度の時間がかかる。空きリストが大きい場合はさらに探索に必要な時間が増える。もう 1 つは、大きな空き領域が分割されずに保存される反面、使われることのない小さな空き領域を数多く作り出す傾向を持つことである。例えば、10byte の領域に 8byte のオブジェクトを割り当てると、2byte の領域が余る。しかし、一般的にオブジェクトの大きさは 4byte 以上であるため、2byte の領域は使われない。

空きリストの探索で必要とする時間を短縮するには、空き領域の大きさを何段階かに区分し、それぞれに応じて別々の空きリストを作って管理する方法も考えられる。また、再利用性の少ない小さな空き領域を発生する欠点に対して、Worst fit 法が提案されている。

Next fit 法

Next fit 法は、First fit 法の空き領域の最初の部分に小さな空き領域が増加する傾向があり、条件に合う領域が見つかるまでの平均時間が増加するという欠点を克服するために提案された方法である。

方法として、空きリストの開始点を、最後に割り当てられたオブジェクトの次に設定する。これにより、大きな領域が残りにくいので、記憶領域の利用効率がやや低くなるが、

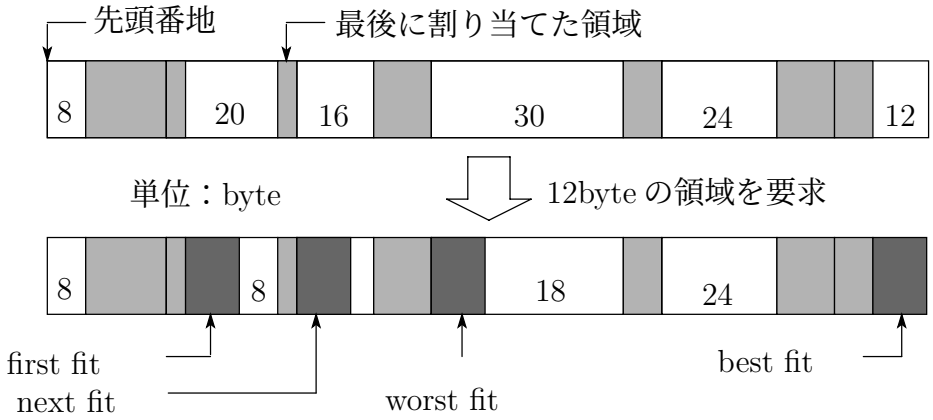

図 2.1: 各 Fit 法の空き領域の探索方法

比較的大きな領域が空きリストに分散するため、割り当てに要する時間が短縮する。

Worst fit 法

Worst fit 法は、Bestt fit 法の欠点である、使われることのない領域の発生を減少させるために提案された割り当て方法である。

方法として、大きさ n の領域が必要な場合、空きリストを順に全てたどり、n 以上で、かつ最も大きな空き領域にオブジェクトを割り当てる。

これにより、Best fit 法の欠点を部分的に克服できるが、Best fit 法と同様に空きリストを全てたどるので、時間がかかることや、大きな領域から切り出すため、大きな空き領域が保存されないという欠点をもつ。

まとめ

本節では Fit 法の代表的な 4 つの割り当て方法について説明した。各割り当て方法の例を、図 2.1 に示す。上段のメモリの状態は、白色の領域が空き領域、灰色の領域が使用領域を表している。

本節では 4 つの割り当て方法について説明したが、どの Fit 法が最も優れているかは、一概には言えない。一般的に、Best fit 法が最も優れていると予測されるが、Best fit 法は、割り当てに使うことができない、小さな領域を多数発生させる傾向を持つため、First fit 法や Next fit 法よりメモリの利用効率が悪いという結果もある [5]。

2.2 Two-level allocation

Two-level allocation は、Boehm-Demers-Weiser コレクタ [7] によって使用された。この方法は、Conservative GC などのコンパクションができない環境で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 階層にしている。

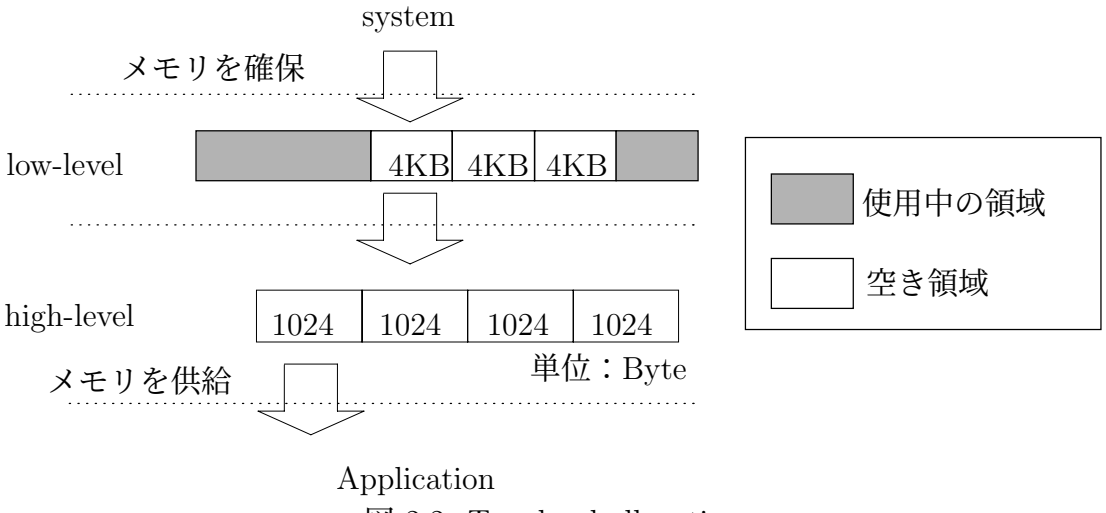

図 2.2: Two-level allocation

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。上位レベルでは、各々の空き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をある固定の大きさに分割して、オブジェクトを割り当てる。このとき上位レベルでは、異なる大きさごとに空きリストを管理する。

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 4KB) 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の大きさ (例えば 8byte、16byte、32byte、…、2KB) を決めて、メモリを供給する。例えば 800byte の領域を要求された場合、図 2.2 のように、下位レベルで確保された 4KB の領域を 800byte に近い 2^nbyte、つまり 1024byte の大きさに区切ってメモリを供給する (残りの 224byte は使わない：内部断片化)。

もしガベージコレクションのスイープフェーズで、あるブロックが全て空だと確認されたら、下位レベルのアロケータに返すことができる。

この割り当て方法は、ブロックをある固定の大きさに分割するため、内部断片化が発生する。このため、Fit 法と比べるとメモリの利用効率は内部断片化の分、低くなる。しかし、オブジェクトを割り当てる際、例えば 16byte の領域は 9〜16byte の要求のみに対応というように固定し、余った領域をフリーリストに加えないことで、Fit 法に比べメモリの再利用性を高めている。なぜならば、Fit 法のように比較的大きな領域が細分化されることなく、オブジェクトが解放された際に 16byte の領域に戻ることができるからである。したがって、外部断片化を軽減できる。さらに、異なる大きさごとに空きリストを管理するため、メモリの割り当て速度が速くなる。

また、キャッシュを備えているコンピュータの場合、下位レベルで獲得するブロックの大きさをキャッシュの 1 ページの大きさと等しい大きさに設定すると、局所性の向上が期待できる。なぜなら、キャッシュの 1 ページよりも小さなオブジェクトがページをまたがらないからである。また、Two level allocation では同じ大きさのオブジェクトがまとまって近くに置かれるので、同じ大きさのオブジェクトが連続して呼び出される傾向がある場

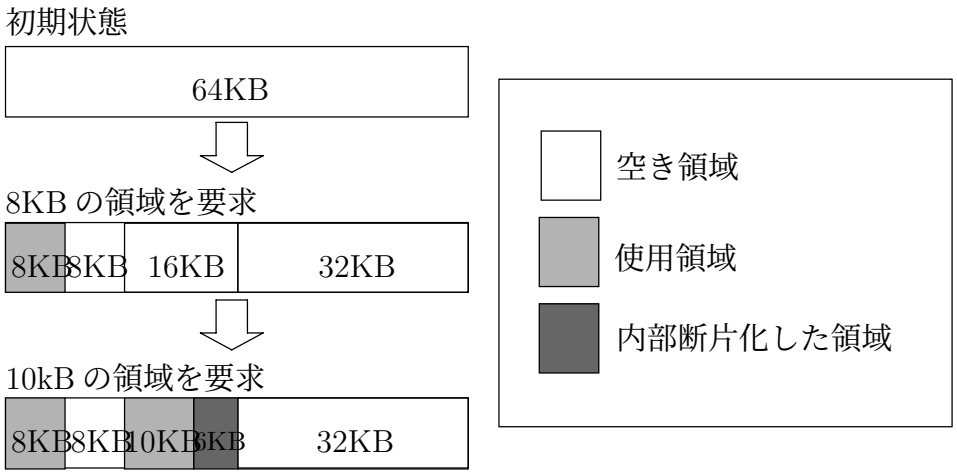

図 2.3: Binary buddy system

合に有効である。なぜなら、ページフォルトの発生する可能性が低いからである。

2.3 Buddy system

Buddy system は Two-level allocation と同様に、メモリの断片化を軽減するために提案された割り当て方法である。最も単純な Buddy system は Binary buddy system[3] と呼ばれている。その他に、Fibonacci buddy、Weighted buddy 、Double buddy system などが提案されている [1]。

2.3.1 Binary buddy system

Binary buddy system は、割り当てる記憶領域の大きさを 2 のべき乗に限定する。要求されるオブジェクトの大きさが 2 のべき乗でない場合は、それより大きい次の 2 のべき乗の領域に余りの領域を含めて割り当てる。

例えば、初期状態では大きさ 2^m の記憶領域が 1 つ存在すると仮定する。その後、大きさ $2^k(0 \leq k \leq m)$ の領域が要求された場合、2^k の領域が存在しないなら、2^k よりも大きな領域を 2 分割することで 2^k の領域をつくる。ある領域が 2 分割された場合、分割された 2 つの領域を分身 (Buddy) と呼び、もし両方の領域が空だと確認されたら、融合してもとの大きさの領域に戻る。つまり、ブロックはペアで処理される。例えば、初期状態が 64KB の領域から 8KB の領域と 10KB の領域が連続して要求された場合を、図 2.3 に示す。

この方法が実用上有効であるのは、ある領域が割り当てられた場合、そのアドレスと大きさが分かれば、その領域の分身のアドレスを導くことができるからである。分身のアド

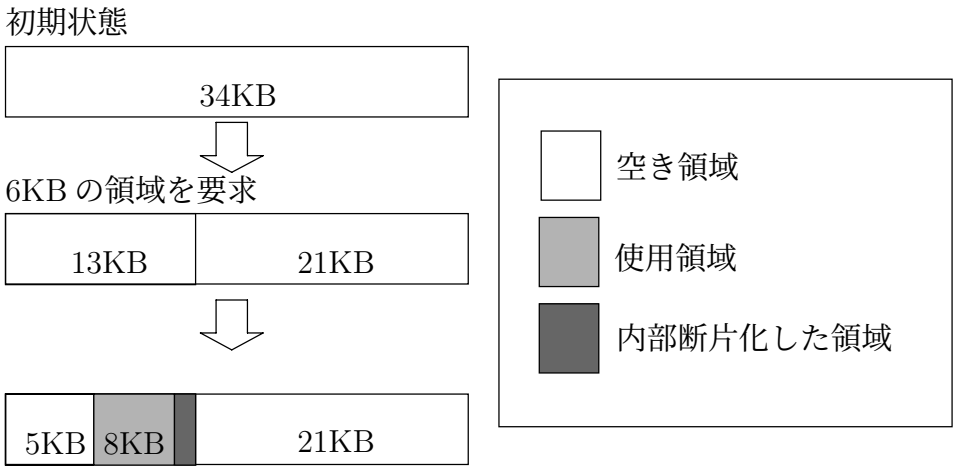

図 2.4: Fibonacci buddy system

レスは、ある領域のアドレスと大きさの排他的論理和を計算することで簡単に求めることができる。例えば、図 2.3 の 10KB の領域が解放された場合、16K(番地) と 16K(大きさ) の排他的論理和をとれば 0(番地) が求まる。

Binary Buddy system は Two-level allocation と同様に、Fit 法と比較するとメモリの使用効率がやや低くなるが、外部断片化を軽減できる。また、異なる大きさごとに空きリストを管理するため、メモリの割り当て速度が速くなる。

2.3.2 Fibonacci buddy system

Fibonacci buddy system は、内部断片化を減少させるために提案された割り当て方法である。Binary buddy system と異なるのは、フィボナッチ数列を用いることで、Binary buddy system のブロックサイズの種類よりも多くのブロックの種類をもつ。このため、Binary buddy sytem と比較すると内部断片化をより減少させることができる。

具体的には、Binary buddy system が $1, 2, 4, 8, 16, 32, \cdots$ という大きさのブロック集合をもつのに対して、Fibonacci buddy system は $1, 1, 2, 3, 5, 8, 13, 21, 34 \cdots$ というブロック集合をもつ。例えば、初期状態が 34KB の領域から 6KB の領域が要求される場合、図 2.4 のように切り出される。

しかし、Fibonacci buddy system は、両方の分身 (buddy) が結合するときに問題がある。Binary buddy system ではアドレスと大きさの排他的論理和で分身のアドレスを算出することができたが、Fibonacci buddy system では分身を計算で算出することができない。

これを解消するために、左分身カウンタ (Left buddy counter) を用いる。方法としては、ブロックが連続で何回左分身の一部になっているのかを示す。例えば、図 2.4 では、5KB のブロックの左分身カウンタは 3 で、8KB、21KB のブロックの左分身カウンタは 0 であ

る。左分身カウンタが 1 以上であれば、その領域の右側に必ず分身が存在する。例えば、図 2.4 の 5KB の領域の分身は、0 (番地) と 5K(大きさ) の和で簡単に分身の番地が求まる。逆に、左分身カウンタが 0 であれば、その領域の左側に必ず分身が存在する。これにより、左右の分身を正しく結合することが可能となる。結合した際、左分身カウンタは 1 減らす。しかし、余分な記憶領域を消費するという欠点をもつ。

2.3.3 Weighted buddy system

Weighted buddy system[4] も、内部断片化を減少させるために提案された割り当て方法で 2 通りにブロックを分割できるという特徴をもつ。初期状態では例えば、3×2^n の大きさの領域をもつ。3 の倍数の大きさのブロックは、例えば、24byte のブロックは 2 つの 12byte のブロックに分割することも可能であり、8byte と 16byte の 2 種類のブロックにも分割できる。2 のべき乗の大きさのブロックは Binary buddy system と同様の分割をする。従って、この割り当て方法は $2, 3, 4, 6, 8, 12, 16, 24, \cdots, 2^n, 3^{n-1}$ という種類のブロックの大きさをもつ。従って、Binary buddy system と Fibonacci buddy system よりも多くのブロック集合をもつ。

Fibonacci buddy sytem と同様に左分身カウンタを設けなければならないが、次節で説明する Double buddy sytem と比べ、どちらかのシステムが枯渇して大きな外部断片化が発生するということがない。

2.3.4 Double buddy system

Double buddy system は、異なる大きさのブロック集合をもつ Binary buddy system を用いる技術で、内部断片化を減少させるために提案された。

例えば以下のように、1 つめの Buddy system は 2 のべき乗の大きさに限定し、もう 1 つは 2 のべき乗の 3 倍の大きさに限定する (m, n は自然数)。

i) $2, 4, 8, 16, 32, 64, 128, 256, \cdots 2^m$
ii) $3, 6, 12, 24, 48, 96, 196, \cdots 3 \times 2^{n-1}$

この例では、Weighted Buddy system と同じだけのブロック集合をもつが、分割の方法が異なる。ブロックは Binary buddy system と同様に半分に 2 分割されるだけで、ある大きさのブロックが要求された場合、最も内部断片化の少ないブロックをどちらかのブロック集合から選ぶ。従って、大きさ 8 のブロックが要求されたが、i) のブロック集合から領域を確保できない場合、例えば、ii) のブロック集合から大きさ 24 のブロックを大きさ 8 と $4(8 \times 2 + 4 \times 2)$ のブロックに分割することはできない。

この方法は、Binary buddy system と比較すると内部断片化をおおよそ半分に減少させることができる。しかし、外部断片化を引き起こす可能性がある。なぜなら、i) のシステムに空き領域が存在しても、ii) のシステムに空き領域がなければ割り当てができない可

表 2.1: 一般的なアロケータの性能測定　[10] より引用

	SVR4	McKusick-Karels	スラブアロケータ
割り当てと解放の平均所要時間 (μsec)	9.4	4.1	3.8
断片化の割合	46 %	45 %	14 %
Kenbus ベンチマークの結果 (1 分間あたりのスクリプト実行数)	199	205	233

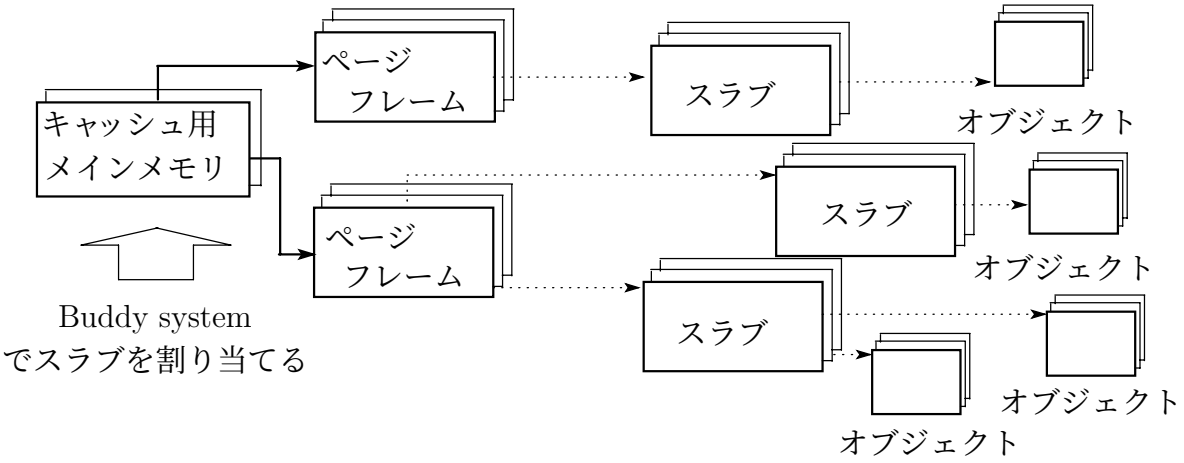

図 2.5: スラブアロケータの構成

能性があるからである。改善策として、比較的大きな空き領域 (例えば、ページの大きさ) を他方のシステムに貸し出すことも考えられるが、処理が複雑になる。

2.4　スラブアロケーション

本節では、Linux2.2 で実際に使われている、スラブアロケータ [8][9][10] について説明する。この方法は断片化の可能性を減少と、処理効率の向上を目的としている。スラブアロケータは Buddy system に基づくメモリ割り当て方法で、Sun Microsystems 社の Solaris 2.4 用に開発された割り当て方法である。この割り当て方法を適用することにより、Buddy system を取り入れていた Linux2.0 よりも、効率が飛躍的に向上した [8]。具体的には表 2.1 のよう測定結果がある [10]。この方法は以下のことを前提としている。

- 格納されるデータの種類により、メモリ領域の割り当て方法を変更する。初期化処理の繰り返しを避けるため、スラブアロケータは、一度割り当てられたオブジェクトを破棄せず、解放されてもメモリ内に残す。スラブアロケータを導入する最大の理由は、Buddy system によるアロケータの呼び出し回数を抑えることである。

- カーネルの関数は、同じ種類のメモリ領域を繰り返し要求する傾向がある。例えば、

新しいプロセスの生成時には、必ず同じサイズのテーブルを割り当てる。よって、割り当てと解放を繰り返すのではなく、キャッシュに蓄えることによって、再利用性を高めている。

- 頻繁に要求が予想される大きさに対しては、その大きさちょうどの領域を複数作成し、要求頻度が低い大きさに対しては、内部断片化の発生は承知の上で、2 のべき乗の大きさで割り当てる。

スラブアロケータはオブジェクトをまとめてキャッシュする。キャッシュは同じ種類のオブジェクト用の置き場所である。例えば、ファイルがオープンされると対応する「open file」オブジェクト用のメモリ領域が必要となる。

キャッシュ用のメインメモリ領域は、複数のスラブに分割される。図 2.5 に示すように、各スラブには、1 つ以上の連続したページフレームがあり、そこには使用中のオブジェクトも未使用のオブジェクトも含まれている。また、スラブアロケータが新しくスラブを作る場合、連続する空きページフレームの組を得るには Buddy system を利用する。

スラブアロケータは、スラブが空きになってもそのページフレームを解放しない。なぜなら、次に空き領域がいつ必要になるか予測不可能であり、空きメモリがまだ多い状況で、オブジェクトを解放しても得るものがないからである。そこで、解放処理を実行するのは、カーネルが空きページフレームを新たに探す場合だけにしている。

2.4.1 議論

本節で説明した、スラブアロケータ以外のアルゴリズムは汎用性をもつ割り当て方法である。つまり、実行するアプリケーションによって、記憶領域の利用効率は違う。

しかし、本研究では組み込み機器を対象としている。よって、スラブアロケーションのように、対象とする環境によってチューニングを施すことも選択肢の 1 つであると考える。なぜならば、組み込み機器では単一のプログラムのみが動作する場合も考えられることや、汎用性を持たせるよりもある程度の専用性をもたせた方が性能が高くなると考えるからである。

次章では、組み込み機器に適していると考えられる割り当て方法について提案する。

第3章　本研究で提案するメモリ割り当て方法

この章では、本研究で提案するメモリ割り当て方法の説明をする。Triple buddy system は、内部断片化を減少させるために Double buddy system を拡張した方法である。また、Block buddy system は、Two-level allocation と Buddy system を組み合わせた割り当て方法である。Separate first fit 法は、記憶領域を分割し、割り当てるオブジェクトの大きさの範囲を決め、First fit 法で割り当てる方法である。

3.1　Triple buddy system

Double buddy system の存在から、3 つ以上の異なる大きさのブロック集合を組み合わせた Buddy system が考えられるのは自然である。複数の Binary buddy system を組み合わせることにより、内部断片化の減少が期待できるという利点をもつ。しかし、異なるシステムが記憶領域を共有するため、あるシステムが枯渇すると、他のシステムで外部断片化の発生する可能性が高い。

しかし、動作させるプログラムによっては、チューニングを適切に施すことにより外部断片化を増加させることなく、内部断片化を減少させることができる。しかし、オブジェクトを割り当てるとき、時間軸に対してオブジェクトの大きさに偏りがある場合には、複数の Buddy system を使用することは逆に記憶領域の利用効率を低下させてしまう。よって、Buddy system を複数使うのは、割り当てるオブジェクトの大きさの分布が一様な場合に適している。

また、3 つ以上の異なるシステムを施した Buddy system に関する文献は見当たらないため、本研究では Triple、Quadruple buddy system について、有効性の検証を行う。

3.2　Block buddy system

Two-level allocation は、大きさ n のオブジェクトを割り当てる場合、ブロックを n 毎に分割して割り当てる。ブロックの大きさが 4KB の場合、例えば表 3.1 のようにブロックが分割される。

表 3.1 を見ると分かるように、8byte 毎に 4KB のブロックを分割した場合 512 個もの

```
free_list_1 = 0 to align_buddy_1
free_list_2 = align_buddy_2 to align_buddy_3
free_list_3 = align_buddy_2 to align_first_fit
free_list_first_fit = bound_free_list to memory_size

allocate() =
   if (bound > object_size)
       if object_size is nearest 2^i
           newcell = free_list_1
           free_list_1 = next(free_list_1)
           return newcell
       if object_size is nearest 3*2^(j-1)
           newcell = free_list_2
           free_list_2 = next(free_list_2)
           return newcell
       if object_size is nearest 5*2^(k-1)
           newcell = free_list_3
           free_list_3 = next(free_list_3)
           return newcell
   else
           newcell = free_list_first_fit
           free_list_first_fit = next(free_list_first_fit)
           return newacell
```

図 3.1: Algorithm: Triple buddy system

表 3.1: Two-level allocation の大きさと個数の関係

大きさ	8byte	16byte	32byte	64byte	128byte	···	2KB
個数	512	256	128	64	32	···	2

```
free_list_1
block_list = 0 to align
        free_list_first_fit = align to memory_size
        allocate() =
            if (align > object_size)
                 newcell = free_list_1
                 free_list_1 = next(free_list_1)
                 return newcell
            else
                newcell = free_list_first_fit
                free_list_first_fit = next(free_list_first_fit)
                return newacell
```

図 3.2: Algorithm: Block buddy system

8byte 単位の領域を確保することになる。したがって、8byte の大きさの割り当て要求が少ない場合、メモリの使用効率は下がってしまう。また、大きなオブジェクトほどブロック内の個数が少ないため、ブロックが解放されると、他の大きさのオブジェクトのためにそのブロックが使用される可能性が高い。小さいオブジェクトならば、そのオブジェクトよりも大きなオブジェクトを扱うブロックに割り当てることもできるが、大きなオブジェクトでは難しい。よって、比較的大きなオブジェクトのための領域を保存しておくことが、外部断片化の増加を防ぐことができると本研究では予測する。

本研究で提案する Block buddy system は、この問題に対処するため、例えば 2kbyte を 50 個、4kbyte を 50 個というように個数を固定し、ブロックで扱う。この領域を確保するために、ブロックを Buddy system でサポートすると効率的に割り当てることができる (図 3.3)。しかし、単純に Binary buddy system を適用する場合は、分割する大きさを 2 のべき乗にしなければならないが、この方法は扱うオブジェクトの大きさが大きいブロックの利用頻度が低い場合にメモリの使用効率は低くなる。よって、ある大きさ以上のオブジェクトは例えば First fit 法などの他の割り当て方法でサポートする必要がある。

また、キャッシュを備えている場合は Two-level allocation と比較すると、局所性が低下する。しかし、本研究で対象としている組み込み機器はキャッシュを備えていない場合が多いため、Two-level allocation よりも有効であると考える。

3.3 Separate first fit 法

本研究で提案するもう 1 つの割り当て方法は、 Separate first fit 法である。この割り当て方法は、全体の記憶領域を扱うオブジェクトの大きさごとに、いくつかの領域に分割

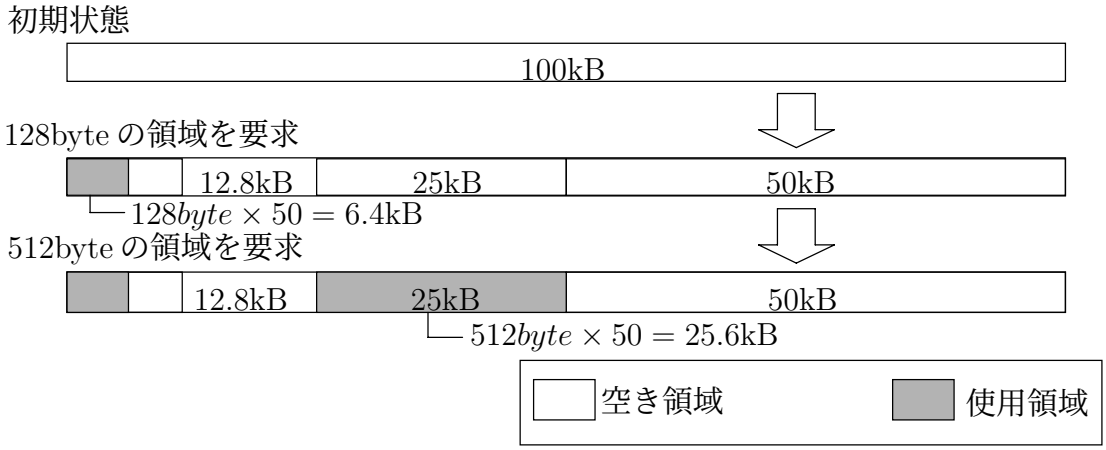

図 3.3: Block buddy system

し、First fit 法で割り当てをする。この方法が有効だと主張するのは、次章以降で説明するシミュレータで各割り当て方法について実験した際に、図 3.5、図 3.6 の状態を得たからである。図 3.5 は First fit 法を実験したときに得られた状態であり、図 3.6 は Two-level allcoation を実験したときに得られた状態である。Two-level-allocation の実験では記憶領域を 2 分割して、8byte から 128byte までのオブジェクトを Two-level allocation で扱い、132byte 以上のオブジェクトを First fit 法で割り当てる。ここで、下段の状態に注目すると空き領域がまとまった大きな領域であることが分かる。しかし、First fit 法の実験で得た状態を見ると、大きな領域が少ない。つまり、First fit 法では外部断片化が進んでいるのに対して、Two-level allocation では大きな領域が残っており、外部断片化が少ない。

本研究で問題視している外部断片化は、空き領域が散らばることによって大きな領域を確保できないことが原因である。しかし、この実験結果によると、割り当てるオブジェクトの範囲を決定することにより、空き領域の散らばりを抑えることができることが分かった。したがって本研究では、小さなオブジェクトの要求が少ないアプリケーションには Separate first fit 法が断片化の抑制するのに有効な場合があると考える。 この方法は例えば図 3.7 のように全体の記憶領域を 2 分割し、一方に 0 から 512KB のオブジェクトを割り当て、もう一方には 512KB 以上のオブジェクトの割り当てを行う。

この割り当て方法は、内部断片化は 0 であり、割り当てるオブジェクトの大きさの範囲を決定することで、外部断片化を減少させることができる。なぜなら、一般的な Fit 法に比べ、記憶領域に割り当てるオブジェクトの範囲を決定するため、オブジェクトの大きさに違いが少なくなるからである。

また、フリーリストを複数用いるため、高速に割り当てをすることができる。しかし、割り当てられない場合に、他の領域で割り当てをしないので、分割する境界の位置を誤ればメモリの利用効率は低下する。

```
free_list_1 = 0 to align
free_list_2 = align to memory_size
allocate() =
    if (bound > object_size)
            newcell = free_list_1
            free_list_1 = next(free_list_1)
            return newcell
    else
            newcell = free_list_2
            free_list_2 = next(free_list_2)
            return newacell
```

図 3.4: Algorithm: Separate first fit 法

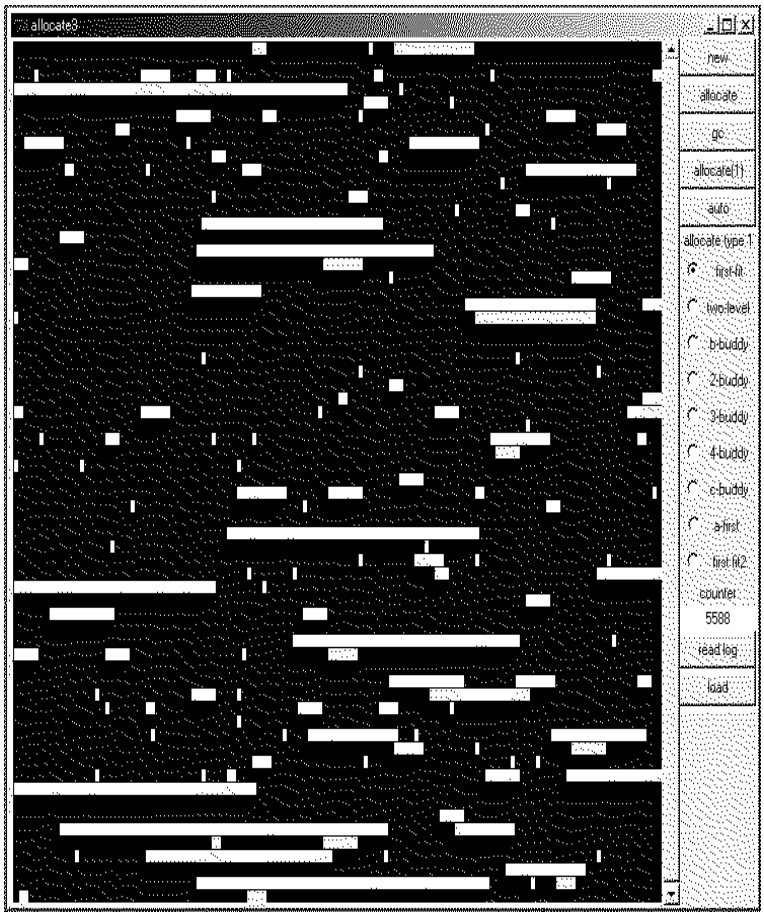

図 3.5: First fit 法のシミュレーション

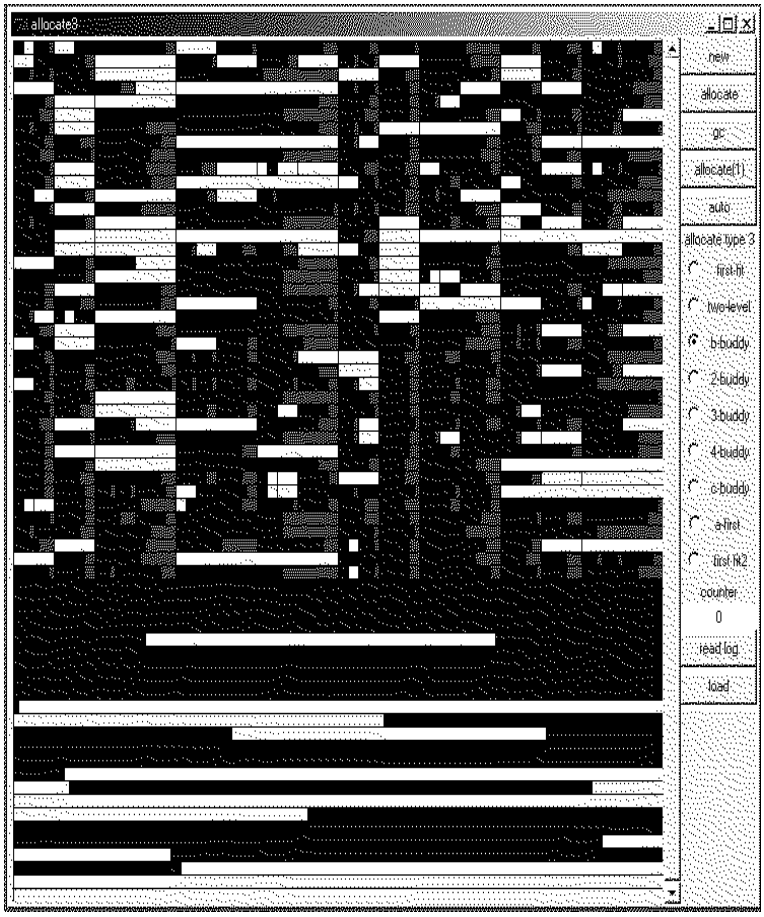

図 3.6: Two-level allocation のシミュレーション

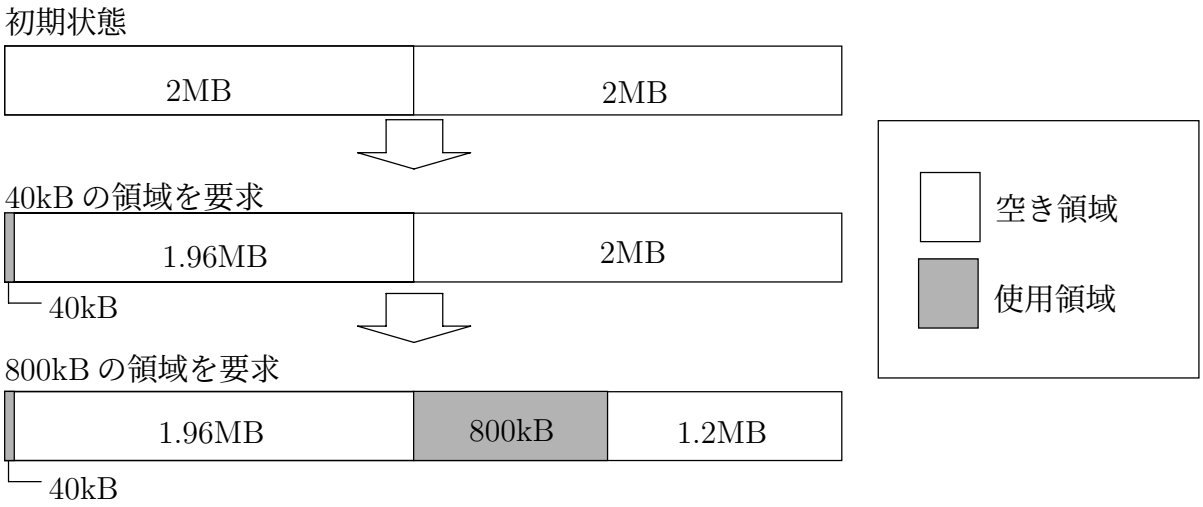

図 3.7: Separate first fit 法

第4章　テスト環境の構築

本章では、各割り当て方法について断片化の状態を測定するテスト環境について説明する。テスト環境を構築するには、以下の 2 通りの設計方法が考えられる。

- KVM の割り当て方法を変更して実験する

利点：KVM の割り当て方法を変更して、実際のアプリケーションを動作させるため、測定結果の信頼性が高い。

欠点：他の小型 Java VM や他の言語仕様の断片化の測定ができない。テスト環境の実装が、乱数を用いた実験に比べて複雑である。また、ユーザからの入力を待つようなアプリケーションの場合、適した割り当て方法の自動算出が難しい。

- 乱数を用いて KVM から離れた環境で実験する

利点：KVM だけでなく、他の小型 Java VM や他の言語仕様の断片化を測定することができる。テスト環境の実装が比較的簡単である。

欠点：乱数を用いて実験をするため、測定では平均的な結果をだす。

本来ならば、様々な種類のアプリケーションを様々な割り当て方法で実行し、断片化の可能性が減少するようにチューニングすることが、測定結果の信頼性が高いため、好ましい。しかし、KVM 上で動作するプログラムのほとんどはゲームプログラムであり、メモリの状態を測定するとほとんど断片化することはない。従って、どちらの設計方法も大切であるが、本研究では乱数を用いたテスト環境を構築する。

断片化が生じない原因は、対象機器がリソースに限りのある組み込み機器であるため、単純なゲームプログラムが多いことと、プログラマがメモリ管理に慎重であることが原因であると考えられる。

4.1　要求仕様

テスト環境に必要であると考える事項を以下で説明する。

1. **パラメータの変更が容易であること**
   最も適した、割り当て方法を探し出すため、ブロックのサイズやブロック内に確保するオブジェクトの数、記憶領域を分割する位置、扱うオブジェクトの大きさの範囲、記憶領域の大きさ、Buddy system での記憶領域を分割する位置などのパラメータを

何度も変更しなければならない。よって、パラメータの変更が容易であることが、実験に必要な時間の短縮につながる。

2. **視覚的に断片化の様子を見られること**
   断片化の状態を測定する際、理論や数値を見てパラメータを変更するとともに、断片化の状態を視覚的に見るため、メモリ内部の空き領域、使用領域の状態をグラフィカルに表示することで、感覚的にパラメータを変更することも必要である。
3. **自動的に、適切な割り当て方法を導くこと**

4.1.1 構築したテスト環境の機能

本研究で構築したテスト環境の主な機能を以下に示す。残念ながら、自動的に適切な割り当て方法を導くことはできていない。

- 視覚的にメモリ内部の様子を見ることができる。断片化の状態を判断するのは難しいた。しかし、この機能を使うことにで、使用領域、空き領域、内部断片化の状態を見て人間が感覚的にパラメータや割り当て方法を操作することが可能となる。
- 実アプリケーションでのメモリの状態を視覚的に見ることができる。しかし、実アプリケーションの割り当て · 解放のログファイルをテスト環境に読み込ませることで実現しているので、KVM と連動していない。
- リストを読み込むことで、表 4.2 の内容を変更することができる。
- 割り当て方法や、パラメータを容易に変更できる。ただし、実行途中に変更はできない。

今回作成したテスト環境を、図 4.1 に示す。視覚的に断片化の状態を見ることができる。白の領域は空き領域、灰色の領域は内部断片化した領域、黒色の領域は使用領域を表す。視覚的にメモリ内部を見る利点は、直感的な改良を施すことができることである。たとえば、本研究で提案する Separate first fit 法は、シミュレータが視覚的にメモリ内部を見せることによって、気が付くことができた。テスト環境の右側にあるボタンを使うことで、容易に割り当て方法の変更や、リストやログファイルの読み込みをおこなうことができる。また、一定の領域を解放した時点から割り当てをする際、全体でどれだけのオブジェクトの割り当てたのかを示すカウンタを設けた。

4.2 テスト環境の準備

本節では構築したテスト環境を用いて実験する前に、どのように乱数を使って割り当てをするのか説明する。本研究では乱数を用いてテストをおこなうが、まず割り当てるオブジェクトの大きさを決定しなければならない。割り当てるオブジェクトの大きさや出

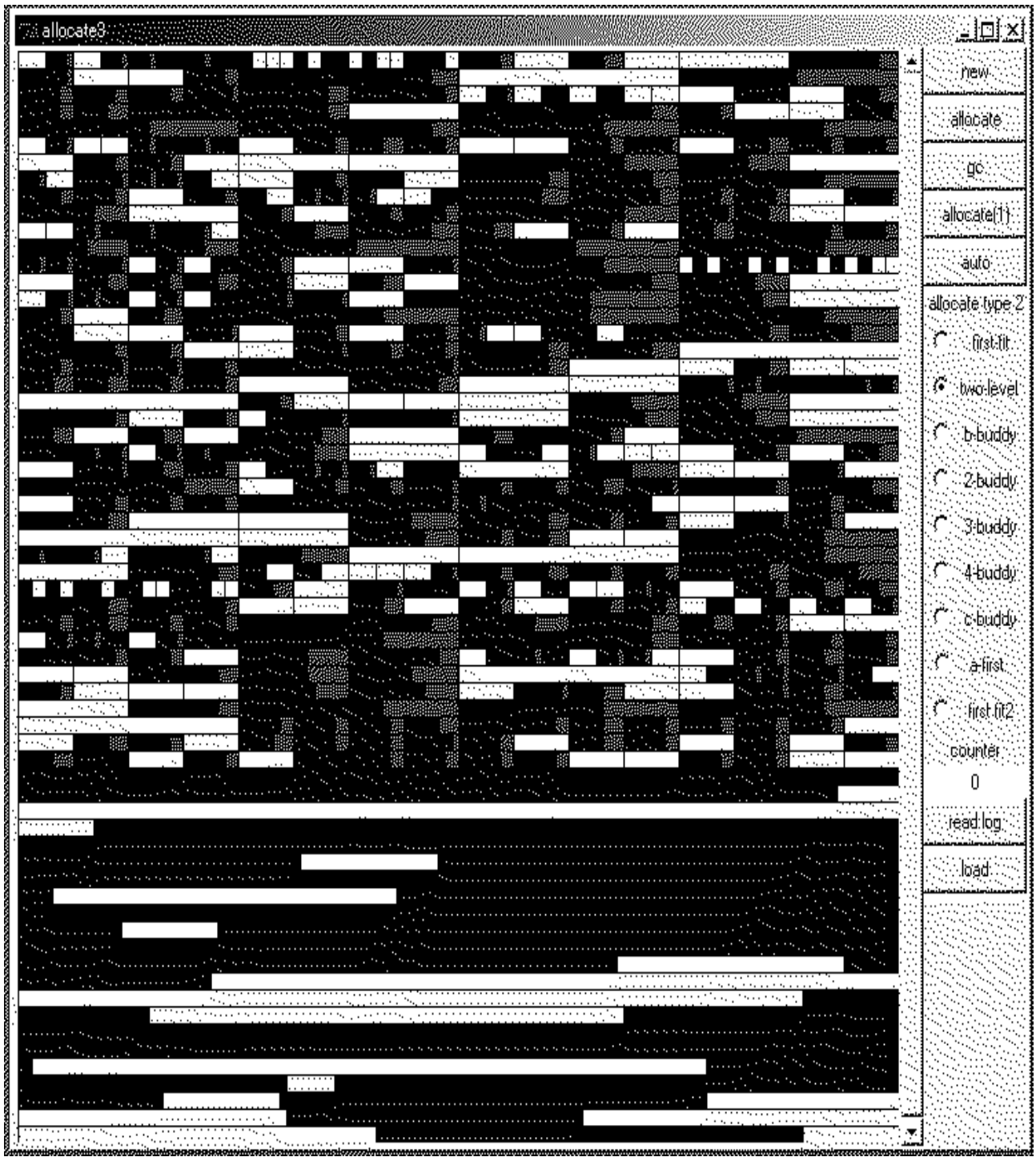
allocate3
new
allocate
gc
allocate(1)
auto
allocate type 2
first-fit
two-level
b-buddy
2-buddy
3-buddy
4-buddy
c-buddy
a-first
first fit2
counter
0
read log
load

図 4.1: テスト環境

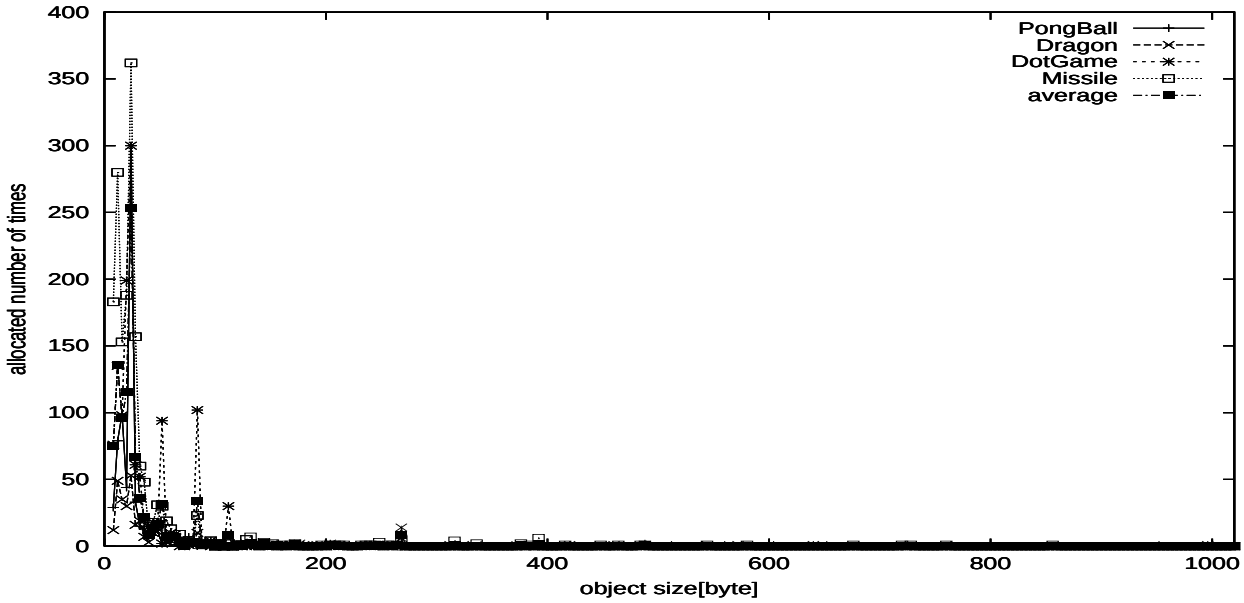

図 4.2: 割り当てられたオブジェクトの大きさと回数の関係

size[byte]	8	12	16	20	24	28	32	36	40	44	48
個数	16	28	20	23	30	13	7	4	2	3	3
size[byte]	52	56	60	64	68	76	84	96	112	124	132
個数	6	2	1	1	1	1	7	1	2	1	1
size[byte]	144	172	200	248	320	392	720	800	1024		
個数	1	1	1	1	1	1	1	1	1		

表 4.1: シミュレータで使用するオブジェクトの大きさと個数の関係

現頻度をできるだけ実機に近い状況をしなければならないので、4 種類のゲームプログラムを実際に実行し、割り当てられたオブジェクトの大きさと回数の統計を得た。これは、各プログラムが 64KB のメモリを消費するまでのデータから抽出している。図 4.2 に示す統計に基づき、表 4.1 の数値を持つリストを作成した。このリストを使い、乱数を用いて表 4.1 中の任意の大きさのオブジェクトを割り当てることにする。もうオブジェクトの割り当てができない状況になれば、任意の割合のオブジェクトを解放する。 Two-level allocation や Buddy system といった、内部断片化の発生を承知する割り当て方法は、一般的にキャッシュの 1 ページ以上の大きさを持つオブジェクトの要求に対して、別の割り当て方法を用いる。なぜなら、大きなオブジェクトの要求まで Two-level allocation などの割り当て方法で対処すると全体の内部断片化がさらに増加するからである。よって、本研究では、ある大きさ以上の要求は First fit 法で割り当てをサポートすることにする。

次に、First fit 法で扱うオブジェクトの大きさを決定しなければならない。図 4.2 を見

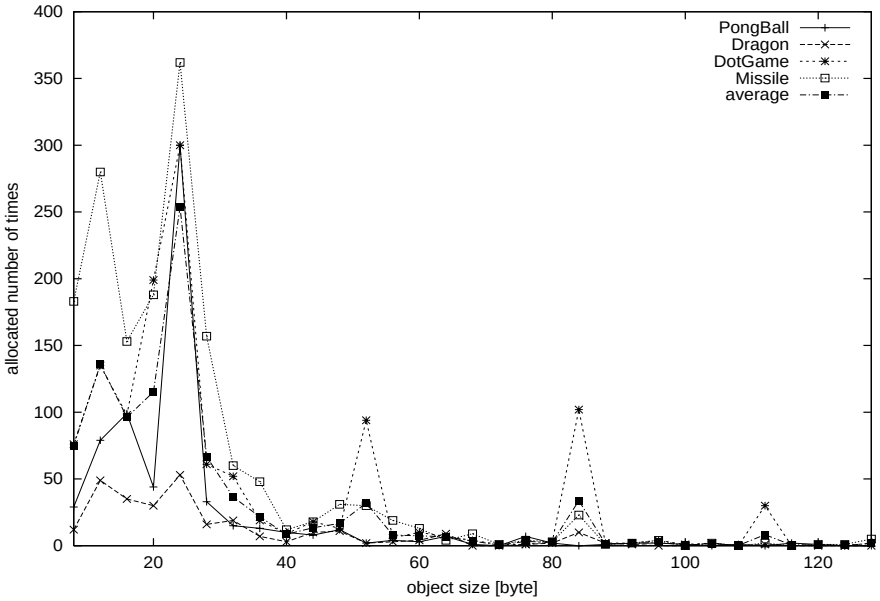

図 4.3: 大きさ 0〜128byte のオブジェクトと回数の関係

ると、大きさが 128byte 程度までのオブジェクトが頻繁に割り当てられていることが分かる。図 4.2 を横軸を 0〜128byte に限定したものが、図 4.3 である。132byte(KVM の割り当ては、8byte 以上 4byte 単位に限定しているので、128byte の次は 132byte になる) 以降の要求頻度の少ないオブジェクトを Two-level allocation などの割り当て方法でサポートすると、内部断片化がさらに増加することは明らかである。よって、本研究では 0〜128byte までのオブジェクトを Two-level allocation や Buddy sysytem で割り当て、132byte 以上のオブジェクトを First fit 法で割り当てることにする。

第5章　断片化の計測

本章では、テスト環境を用いて断片化の状態を計測して得た結果を示すとともに、その考察をおこなう。また、断片化の計測方法について説明する。

5.1　断片化の計測

本節では、断片化の計測方法について説明する。断片化を測定するのに最も一般的に使われる方法は、以下の式で表される。

$$\frac{\text{(実際に割り当てられた領域の総量} - \text{要求されたオブジェクト全ての総量)}}{\text{実際に割り当てられた領域の総量}}$$

例えば、この方法は、Mark S. Johnstone,Paul R. Wilson の断片化を 0 にすることを目的にした研究 [2] でもこの方法の方針をもとに、4 つの方法を提案している。以下にその測定方法とその問題点について述べる。説明の簡単化のため上記の測定方法を A とする。

1. **全てのポイントで、A の計算をし、平均値を算出する**
   問題点：断片化の増減が分からない。
2. **要求されたオブジェクト全ての総量が最も大きなポイントと、それに対応する実際に割り当てられた領域の総量のポイントで A の計算をする**
   問題点：要求されたオブジェクト全ての総量が最も大きなポイントで、偶然メモリの利用効率がよければ、測定結果は良くなる。
3. **実際に割り当てられた領域の総量が最も大きなポイントと、それに対応する要求されたオブジェクト全ての総量のポイントで A の計算をする**
   問題点：実際に割り当てられた領域の総量が最も大きなポイントで、偶然ライブメモリの総量が少なければ、測定結果は非常に悪くなる。
4. **要求されたオブジェクト全ての総量が最も大きなポイントと、実際に割り当てられた領域の総量が最も大きなポイントで A の計算をする**
   問題点：ライブメモリの総量が時間とともに減少すれば、測定結果は良くなる。

この方法では内部断片化を許す割り当て方法の場合、測定結果は良くならず、内部断片化、外部断片化ともに減少させなければならない。本研究では、組み込み機器を対象としているので、外部断片化と内部断片化の総量を減らすことよりも、外部断片化の発生を抑えたいと考えている。なぜなら、ある程の度断片化が発生させてでも、長時間プログラム

が実行可能であることが目的だからである。また、Two-level allocation や Buddy system のように内部断片化を発生させることで、外部断片化を減少できると考えているため、上記の測定方法は本研究には適用できない。

よって、本研究では外部断片化を重点的に反映するような測定方法 2 つ考えた。1 つは、すべてのポイントで A の値をとり、最小自乗法で近似直線を求め、その傾きの大きさで、断片化の測定をする方法である。なぜならば、右肩上がりであるほど再利用性が下がっていることを示すので、外部断片化が発生していると考えられるからである。もう一つは、GC 後の空き領域と、実際に割り当てることのできた領域の差をとり、GC 後の空き領域を 100 %ととして、値を導き、1 つめの方法と同様に傾きで断片化を測定する方法である。以降での説明の簡単化のため、前者の測定方法を B、後者の割り当て方法を C とする。

5.2 実験方法

リストからランダムに割り当て、解放という動作を繰り返すことで、断片化を擬似的に発生させることができる。解放動作で、全体の記憶領域に対してある一定割合の空き領域を確保する。この状態から、全体でどれだけの量のオブジェクトが割り当てられたのかを測定する。この場合、C の測定方法をとれば解放後の領域が一定なので、割り当てた領域の総量だけを測定すればよい。この、割り当て、解放という動作を 100 回繰り返し、その割り当てたオブジェクトの大きさの総量を平均した数値を算出する (図 5.1)。なお、内部断片化の発生する割り当て方法については、内部断片化した領域の大きさは結果に含んでいない。

5.3 各割り当て方法に対するチューニング

各割り当て方法ごとに最適な結果を得るため、各割り当て方法に合った改良を施す。全割り当て方法に共通して、以下の環境を仮定して実験をする。

1. 全体のメモリ領域　　：　32KB と 64KB
2. メモリを解放する割合　　：　30 %と 50 %

5.4 Fit 法

First fit 法についての実験結果を、表 5.1 に示す。First fit 法の結果を見ると、全体の記憶領域が 32KB の場合、30 %の空き領域 (9.6 k B) を確保すると、約 2 割の領域しか割り当てることができない。しかし、50 %の空き領域 (16KB) を確保した場合は約 4 割の領域を割り当てることができる。また、全体の領域が 64KB の場合、30 % (19.2KB) の空き領域を確保したとき約 4 割の領域を、50 %の領域 (32KB) の空き領域を確保したときは

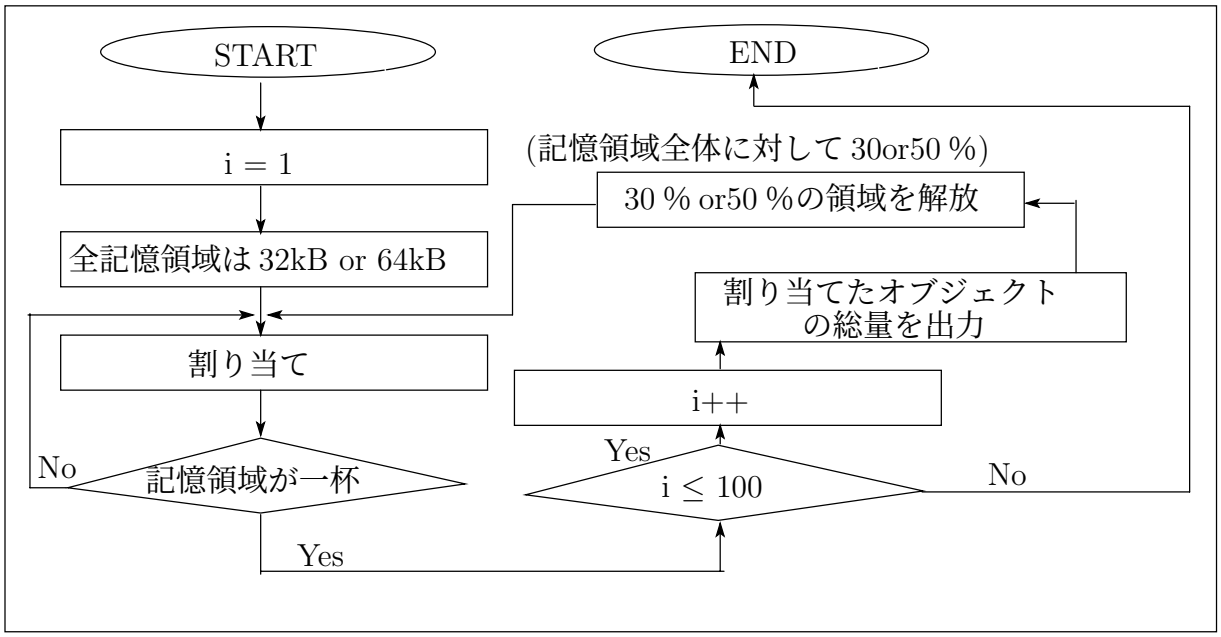

図 5.1: シミュレーションのアルゴリズム

表 5.1: Fit 法のシミュレーション結果

環境	32KB：30 %	32KB：50 %	64KB：30 %	64KB：50 %
Fist fit (byte)	2005	4244	4701	7713
Best fit (byte)	3382	9832	7849	22055

約 42 %の領域を割り当てることができる。空き領域が大きいほど、記憶領域の利用効率が高くなるということは予測しやすいが、この実験結果より再確認できた。

しかし、全体の記憶領域が 32KB の場合、解放する領域の割合を 30 %から 50 %にすることによって利用効率が 2 倍になるのに対し、全体の記憶領域が 64KB の場合は 32KB のときほど顕著な差が現れなかった。

一般的には Best fit 法よりも First fit 法が優れている場合が多いといわれているが、この実験では Best fit 法が明らかに First fit 法に勝っている。このような結果になったのは、シミュレーションで割り当てるオブジェクトは 8byte 以上 4byte 単位 (KVM の割り当て方法) で 1024KB までの大きさで実験しているため、オブジェクトの種類も 31 種類しかない。このため、Best fit 法では、記憶領域の再利用性が高くなったのだと予測される。

ここで、32KB の記憶領域、解放する割合 30 %で、１〜1024KB(4byte 単位) のオブジェクトをランダムに割り当てた。この場合、256 種類の大きさのオブジェクトを持つことになる。表 5.2 の結果を見るとほとんど差がないことがわかる。この場合、全ての空きリストを調べなくてはならない Best fit 法よりも、First fit 法が全体的な性能が良いと考えら

表 5.2: Fit 法のシミュレーション結果 (1〜1024byte)

割り当て方法	First fit 法	Best fit 法
平均値 (byte)	6329	6542

表 5.3: Two-level allocation のシミュレーション結果

環境	32KB：30 %	32KB：50 %	64KB：30 %	64KB：50 %
ブロック: 128byte	3245	8063	10180	19591
ブロック: 256byte	3129	9394	9585	21408
ブロック: 1024byte	1940	7823	9215	18306

れる。これらの結果から、オブジェクトの大きさの種類が少なければ、Best fit 法の性能が非常に高いことが分かった。

5.4.1 Two-level allocation

本節では、Two-level allocation の実験について説明する。実験をする前に、決定すべきことがある。本研究では全体の領域を 2 分割し、一方は Two-level allcoation で割り当て、128byte 以下のオジェクトをサポートする。もう一方は First fit 法で割り当て、132byte 以上のオブジェクトをサポートする。このとき、記憶領域を 2 分割する際の割合を考えなければならない。4.1 から、0〜128byte のオブジェクトの総量は 4776byte、それ以上の大きさのオブジェクトは 4152byte である。しかし、単純に 4776：4152 の比で記憶領域を分割できない。なぜならば、内部断片化の総量を考えなければならないからである。内部断片化を考慮した場合、0〜128byte のオブジェクトの総量は、内部断片化の総量 1554 ＋ 4776=6330byte となる。よって、6330：4152 の比で 32KB の領域を分割すると、19785byte の領域を Two-level allocation がサポートすることになる。ブロックサイズを考慮すると、19712byte が適当となり、First-fit 法で 13056byte の領域をサポートする。

次に Two-level allocation 固有のパラメータとして、ブロックの大きさを決定しなければならない。Two-level allocation では 128byte までのオブジェクトを扱うため、最低 128byte のブロックが必要となる。よって、ブロックサイズ 128byte と 256bytne、1024byte の 3 通りの実験をおこなう (表 5.3)。

実験結果より、ブロックの大きさが 128byte と 256byte の場合を比較しても、ほとんど差がないが、1024byte では記憶領域の利用効率が低くく、特に 32kB：30 %の実験は著しく効率が低下している。ブロックサイズが小さいほどブロックの再利用性が高いため、時間軸に対して割り当てるオブジェクトの大きさに偏りがある場合に対応することができるが、大きいオブジェクトのための領域を保存できなくなる。

表 5.4: Buddy sytem のシミュレーション結果

環境	32KB：30 %	32KB：50 %	64KB：30 %	64KB：50 %
Binary buddy	3327	5856	8238	17644
Double buddy	3456	8316	8521	19642
Triple buddy	3513	8962	9902	20841
Quadruple buddy	3620	9206	9876	21486

5.4.2 Buddy system

本節では、Buddy sytem の実験について説明する。記憶領域を 2 分割する際、Binary buddy system の場合は Two-level allocation と同様の位置に First fit との境界を持つ。しかし、Buddy system の実装では、決められ領域を十分に利用できない可能性がある。なぜならば、普通 Buddy sytem の初期状態は 2^n の大きさをもつからである。Binary buddy sytem がサポートする 19785byte の領域で、2^n の大きさで近似すると 16384byte となり 3401byte が無駄になる。これを解決するのは簡単で、例えば 128byte の大きさの Binary buddy sytem を並べれば良い。これは、プログラム起動時に空きリストを分割しており、不必要に buddy の結合を行わないので処理性能の向上も期待できる。

Binary buddy sytem の実験結果は表 5.4 になった。この結果を見ると、Two-level allocation とそれほど変わらない結果となった。対の領域が両方とも空き領域にると、倍の領域になるため、Two-level allocation よりも記憶領域の再利用性が高いと予測していたが、実験結果には現れなかった。

次に、複数の Binary buddy sytem を使用することを考える。Double buddy system は 2^n、$3 \times 2^{k-1}$ の領域を、Triple buddy sytem は 2^n、$3 \times 2^{k-1}$、$5 \times 2^{l-1}$ の領域を、Quadruple buddy sytem は 2^n、$3 \times 2^{k-1}$、$5 \times 2^{l-1}$、$7 \times 2^{m-1}(k, l, m, n$ は自然数) の領域ををもつ。

実験結果より、異なるオブジェクトの大きさをサポートする Binay Buddy system を複数利用すると、内部断片化が減少させつつ外部断片化を抑えることができる。しかし、使用する Binay buddy system の数が多いほど、あるシステムの記憶領域が枯渇したときの対処が複雑になりうる。よって、時間軸に対して割り当てるオブジェクトの大きさに偏りがあると、記憶領域の利用効率は低下する。

5.4.3 Block buddy system

本節では Block buddy system の実験結果について説明する。Block Buddy system の場合も、サポートする領域は 19785byte であるが、ブロック内部の個数によって、最小の Buddy system の大きさが変わる。例えば、10 個単位でブロックを確保する場合、128byte×10 が最小の Buddy sytem の構成となる。よって、ここでは 19200byte を Block buddy system がサポートする。

ここで、ブロック内部の個数を決定しなければならない。もし、1であるとBinary buddy sytemと同様となる。また、ブロック内部の個数が多いと、逆に記憶領域の利用効率は低くなる。本研究では、5個、10個、20個 (表 5.5) の3種類について実験をした。5個と10個の実験結果はほとんど等しいが、20個の場合は少し記憶領域の利用効率が低く、予測通りの結果を得た。

表 5.5: Block buddy systemのシミュレーション結果

環境	32KB：30 %	32KB：50 %	64KB：30 %	64KB：50 %
個数：5個	3502	9580	9249	21750
個数：10個	3560	9504	9121	20348
個数：20個	2889	9260	7571	19093

5.4.4 Separate first fit法

本節ではSeparate first fit法での実験結果について説明する。Block buddy sytemのサポートする記憶領域は、Two-level allocationやBuddy sytemと異なり、4776：4152の比率で分割する。よって全体の記憶領域が32KBの場合17530byte、64KBの場合35061byteを0〜128byteの大きさおオブジェクトで割り当てることになる。

表 5.6の実験結果を見ると、本章で実験した割り当て方法の中で最も利用効率が高い。この割り当て方法は、他の割り当て方法と比較して最も実装が容易な方法でもある。しかし、Best fit法の実験結果と同様に、割り当てるオブジェクトの種類が少ない場合に有効であると考えられる。よって、割り当てるオブジェクトの種類が増加した場合に、良い性能が出るとは限らない。

表 5.6: Separate first fit法のシミュレーション結果

環境	32KB：30 %	32KB：50 %	64KB：30 %	64KB：50 %
平均値 (byte)	3274	11387	8604	24637

5.5 考察

Two-level allocationやBuddy systemでの結果は内部断片化した部分が含まれていないという側面もあるが、 本章での実験では、Separate first fit法が最も高いという結果を得た。図 5.2、図 5.3、図 5.4に示す、First fit法とSeparate first fit法、Block buddy sysytemの実験データの詳細を見ると、First fit 法が全体的に右肩下がりなのに対し、Separate first

fit 法と Block buddy system は広い範囲で見れば一定である。つまり、First fit 法では割り当て · 解放を繰り返すうちに外部断片化が発生し、外部断片化がヒープ領域を圧迫することによって記憶領域の利用効率が悪化したことを示している。First fit 法以外の割り当て方法は First fit 法、Block Buddy sytem と同様にほぼ一定の値を示した。

一般的に断片化を計るには、実際に割り当てている領域の総量と実際に要求されている領域の総量との差で導くことが多い。しかし、本研究では次のような優先順位で断片化の性能を測定する。

1. **100 回の測定結果をグラフにした際、横軸と平行であること**
   外部断片化の発生が少なく、解放された領域の再利用性が高いことを示している。
2. **グラフの縦揺れが少ないこと**
   グラフの縦揺れが大きいほど、不安定であるといえる。また、平均値や最大値が大きいことよりも最小値が大きいほどよいと考える。なぜならば、平均値が高くても最小値が 0 に近い場合、要求されるオブジェクトを割り当てることができず、アプリケーションの実行が止まる可能性があるからである。
3. **100 回の測定結果をグラフにした際、グラフの縦軸の数値が高いこと**
   グラフの縦軸の数値が高いほど記憶領域の利用効率が高いことを示している。

本研究の実験でこの計測方法を使うと、例えば、64KB の領域で 50 %の領域では Block Buddy system と Separate first fit 法を比べると、以下の表 5.7 のようになり、Separate first fit 法よりも Block buddy system のほうが優れていると判断される。

表 5.7: Block Buddy system と Separate first fit 法の断片化の測定

	Block buddy system	Separate first fit 法
測定 1	○	×
測定 2	○	×
測定 3	×	○

実験結果より、本研究では 1 つの指針として図 5.5 のようにアプリケーションの特性によって割り当て方法を選択すれば良いと提案する。

A: Two level allocation と First fit 法というように、割り当て方法で扱う領域の境界の位置が静的に決まる場合
B: 扱うオブジェクトの分布が、小さいオブジェクトほど多い場合
C: 断片化よりも実行速度を重視したい場合
D: 扱うオブジェクトの分布で、Two-level allocation などの外部断片化の減少を目的とした割り当て方法が扱うオブジェクトの大きさの範囲で、大きなオブジェクトが多い場合

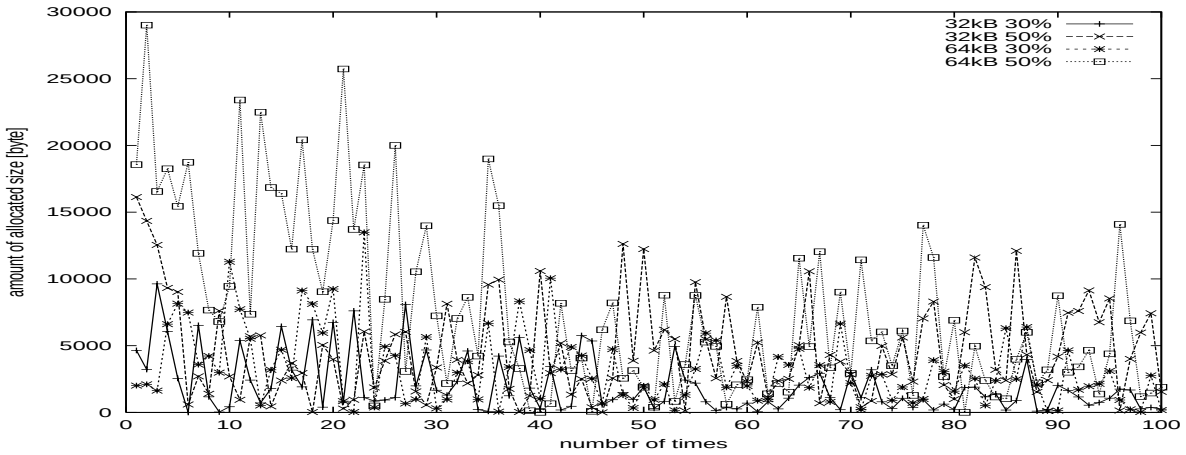

図 5.2: First fit 法のシミュレーション結果

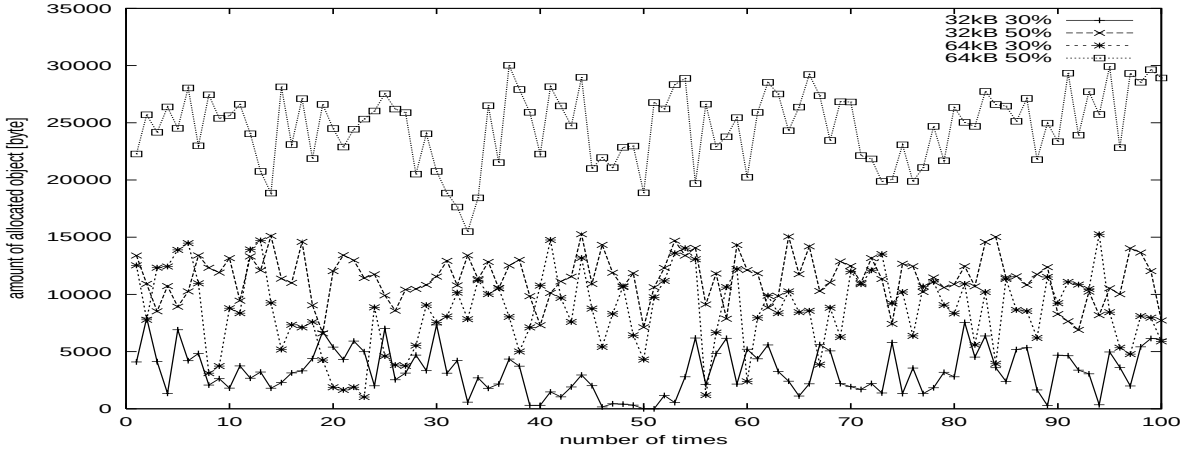

図 5.3: Separate first fit 法のシミュレーション結果

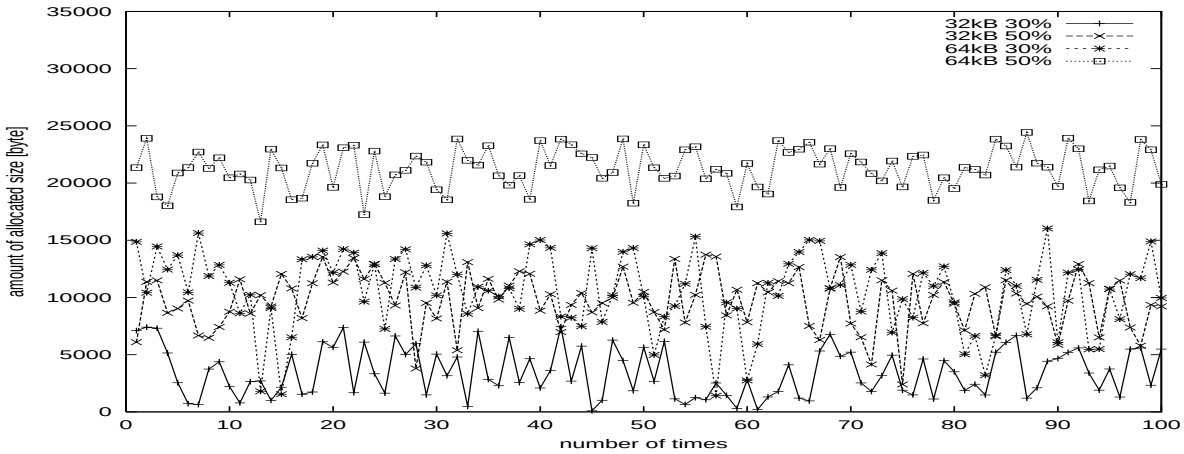

図 5.4: Block buddy system のシミュレーション結果

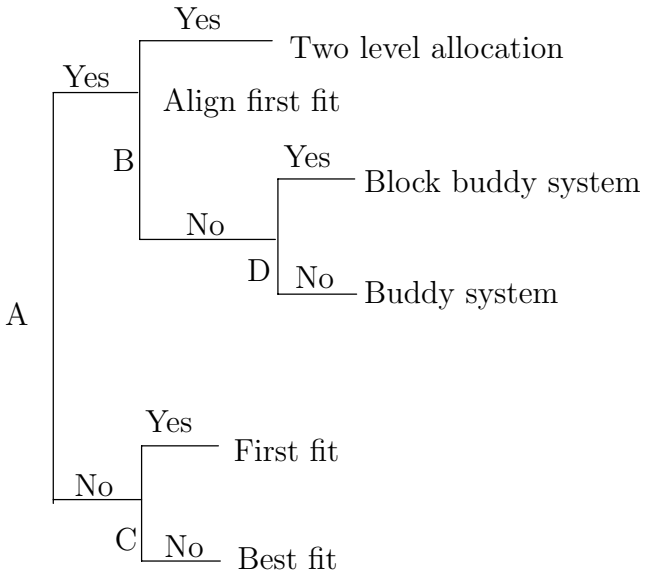

図 5.5: 割り当て方法の選択

しかし、シミュレーションの割り当てに関して、実際のアプリケーションに見られるような、偏りがない。つまり、5byte が連続で 100 回割り当てられるというようなことがない。よって、シミュレーションで得た結果と、KVM 上で実際に動作させた結果とは必ずしも一致しない。また、Two-level allocation と First fit 法とを区切る境界を決定するのもシミュレーションのように容易ではないと考えられる。

第6章　関連研究

本章では関連研究について説明する。

6.1　メモリ管理機能のモジューラかつ効率的な実装

言語処理系に対し、さまざまなメモリ管理システムを実装するためのインタエースを提供する研究として、O'caml を対象とした内山、脇田による研究がある [13]。

メモリ管理システムを実装するためのインタフェースを提供することにより、処理系の実装と独立にメモリ管理システムが実装できる。メモリ管理システムを処理系の他の部分から独立させることは、以下の利点がある。

- 処理系に非依存なメモリ管理システムを提供することで、言語処理系を実装する際のコストを軽減する
- 個々のアプリケーションごとに最適なメモリ管理アルゴリズムを選択することが可能になる。
- メモリ管理アルゴリズムの性能を特定の処理系に依存しない環境で評価できる。

この研究が、メモリ管理システムの実装に有用なものとなるためには、以下の要件を満たさなければならない。

抽象性:処理系とメモリ管理システムとを互いに独立に設計、実装するためには、十分な抽象性を持たなければならない。抽象性の高いメモリ管理インタフェースを提供することで、同一のメモリ管理システムをさまざまな言語処理系で共通に利用することや、同一の言語処理系でさまざまなメモリ管理システムを利用することが可能となる。

効率性:メモリ管理インタフェースは、実装されたメモリ管理システムに対して、既存の処理系でのさまざまな高速化技法を適用でき、期待される性能向上を得られなければならない。このことが可能になると、実用的な処理系として利用することや、メモリ管理システムの性能評価のために利用できる。

しかし、このような抽象的なインタフェースを導入して処理系とメモリ管理システムを独立させると、特定のメモリ管理システムに対して処理系を効率的に実装することが困難となり、処理系の上で動作するプログラムの実行速度を低下させる原因となる。例えば、

同一のアルゴリズムを用いたメモリ管理システムについて、インタフェースを利用して実装したものは、インタフェースを用いずに効率的に実装したものに対して、およそ 10 %の速度低下を示した。

このような性能面の問題を解決する方法として、メモリ管理システムの実装に対して処理系の実装を効率的なものへと変換する手法、および、インタフェース拡張によってメモリ管理システムの効率的な実装を可能にする手法を提案している。これによって、ほぼ性能面の問題を解決している。

言語処理系に対してさまざまなメモリ管理システムを実装するためのインタフェースを提供する他の研究として、*Java Exact VM* (EVM)[14] がある。内山、脇田らの研究との違いは、インタフェースの抽象性にある。EVM が提供するインタフェースは、処理系の開発者を対象としたものであり、VM の実装に対応した具体的なインタフェースとなっている。これに対して、内山、脇田らの研究で提案するインタフェースは、処理系の具体的な実装によらない抽象的なインタフェースであり、処理系の内部の実装に詳しくない者でも、容易にメモリ管理システムを実装できる。

この他には、C++用の拡張可能なメモリ管理ライブラリとしての Customisable Memory Manager (CMM)[15] がある。CMM では、アプリケーションを実装するクラスを継承し、再定義することで、クラスごとに異なるメモリ管理ポリシーを定義できる。

6.2 断片化の減少を目的とした研究

実装に関係なく、様々な割り当て方法の断片化を測定する研究を Mark S. Johnstone、Paul R. Wilson がおこなった [2]。この研究では無駄となる領域を全て排除することを目的としている。

テストプログラムは、Espresso、GCC、Ghostscript、Grobner、Hyper、LRUsim、P2C、Perl の 8 つであり、Sun の SPARCELC 上で計測をしている。

6.2.1 断片化の測定方法

断片化を測定するため、この研究では 4 つの測定方法を提案している。図 6.1 を例にして 4 つの測定方法について説明する。図 6.1 は 2^n 単位でオブジェクトを割り当てる GCC コンパイラのメモリ使用量を表しており、上段のグラフは実際に使用したメモリの総量、下段のグラフは実際に GCC に要求されたメモリの総量を示している。4 つの測定方法について説明するが、簡単化のため、以下のように A と B を取り決める。

A：プログラムに要求されたメモリの総量 (グラフ：下段)
B：実際にアロケータによって割り当てられたメモリの総量 (グラフ：上段)

1. B−A が断片化を示すが、全ての値についてこの計算をし、平均値をとる。B を 100 %とすると図 6.1 では、断片化は 258 %である。

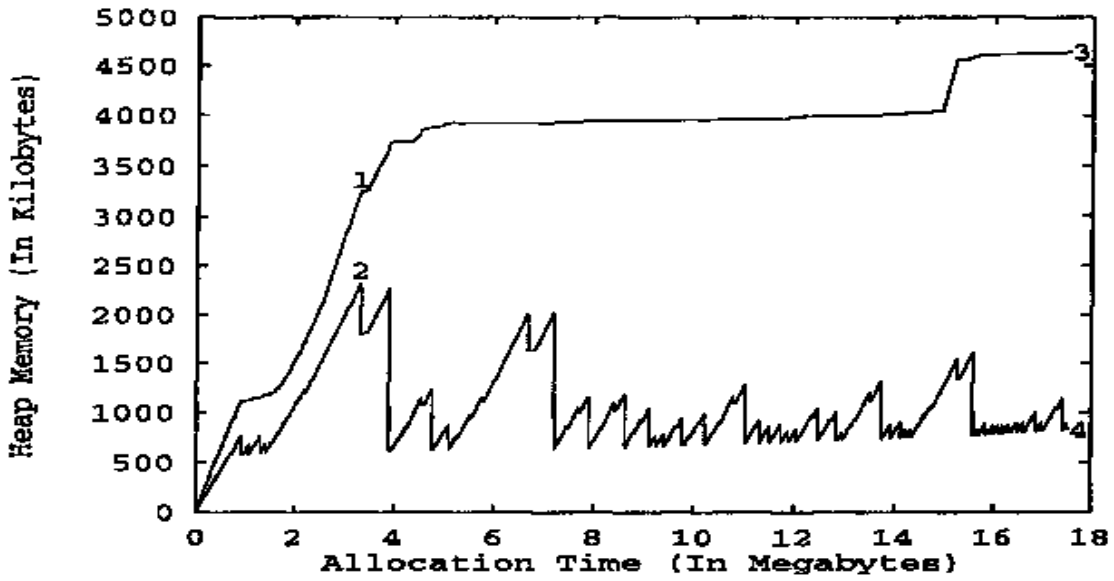

図 6.1: GCC のメモリ使用量

2. B−A が断片化を示すが、(1) と異なり、A の最大値とそれに対応する B の値の差をとる。つまり、図 6.1 の 1 と 2 の差をとる。B を 100 %とすると、断片化は 39.8 %である。
3. B−A が断片化を示すが、(2) と異なり、B の最大値とそれに対応する B の値の差をとる。つまり、図 6.1 の 3 と 4 の差をとる。B を 100 %とすると、断片化は 462 %である。
4. B−A が断片化を示すが、B の最大値と A の最大値の差をとる。つまり、図 6.1 の 3 と 2 の差をとる。B を 100 %とすると、断片化は 100 %である。

この研究では、(3) と (4) の測定方法を使用して、計測をしている。割り当て方法は、First fit 法、Best fit 法、Next fit 法、Binary buddy system、Double buddy system、2^n 単位での割り当て、$3 \cdot 2^n$ 単位の割り当てである。結果として、First fit 法が最も良い性能を示している。

6.3 議論

この章では関連研究について述べた。ここで、他の研究と本研究との違いを比較、検討する。

6.3.1 メモリ管理の変更を目的とした研究

内山、脇田による研究と本研究の違いは、内山、脇田の研究がGCアルゴリズムの性能を評価するのに対し、本研究では割り当てアルゴリズムの性能を評価していることである。また、内山、脇田の研究は、O'caml のみを対象としているので、性能評価が正確である反面、他の言語仕様の性能評価ができない。一方、本研究では、乱数を用いたシミュレーションで性能評価をおこなうため、性能評価の正確さは低いものの、他の言語仕様を評価する際も、1つの指針となる。

6.3.2 断片化の減少を目的とした研究

S. Johnstone、Paul R. Wilso らの研究では、本研究と同様に断片化の減少を目的としてる。しかし、S. Johnstone、Paul R. Wilso らの研究では図 6.1 の2つグラフが重なることを目標としている。よって、内部断片化の発生する割り当て方法は、性能が低いと判断されてしまう。

逆に本研究では、外部断片化の発生する可能性を減少させることを目的としている。よって、内部断片化を発生させたとしても、記憶領域の再利用性を高めるこで、外部断片化が減少すれば性能が良いと判断している。また、この研究がワークステーションを対象にしているのに対し、本研究は組み込み機器を対象にしている

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

本研究をおこなうにあたってメモリ割り当て方法の基本的なアルゴリズム (Fit 法、Two-level allocation、Buddy system)、linux で実際に用いられているスラブアロケーションの解説をした。その後、本研究で提案するメモリ割り当て方法 (Triple buddy system、Block buddy system、Separate first fit 法) について説明し、各割り当て方法を比較するテスト環境の構築をおこなった。また、このテスト環境を用いて、様々な割り当て方法について実験をした。

次に、本研究に関連する研究内容と、本研究で提案するテスト環境の比較をおこない、本研究の有用性を述べた。

本論文では、組み込み機器ではアプリケーションや環境によって割り当て方法の改良、パラメータの調整をすることがメモリを効率的に利用できると主張した。

7.2　結論

本研究では、各割り当て方法を比較するためのテスト環境の構築を行ない、単純なアイデアで、性能の向上が期待できることを示した。本研究を通じて、単純な割り当て方法でも、外部断片化の可能性を減少できる可能性を持つことや、視覚的に各割り当て方法を見ることが、感覚的に各割り当て方法の動作を理解するとともに、直感的な改良を施せることが分かる。

7.3　今後の課題

本研究では、KVM からいったん離れて実験をした。なぜなら、KVM 以外の小型 VM や言語仕様に対しても実験をすることができるからである。また、KVM のメモリ管理部分を変更するよりも、シミュレーションでメモリ管理部分を構築した方が変更が容易である。

しかし、本研究で構築したテスト環境では、乱数を用いるため、割り当てるオブジェクトの大きさに偏りがない。よって、テスト環境での実験結果性が高い割り当て方法であっても実際のシステムで性能が良いとは限らない。よって、より信頼性の高い割り当て方法を導くためには、KVM に連動したテスト環境の構築も必要である。

また、本論文で示した割り当て方法の他に、実験すべき割り当て方法はいくつも存在する。例えば、Separate first fit 法に関して言えば、上位の割り当てを Best fit、下位の割り当てを First fit、または、両方とも Best fit 法で測定するなど、容易に考えられる割り当て方法の測定をしなければならない。アプリケーションに特化した割り当てに関する実験をしなければならない。

7.4 将来の展望

乱数を用いたテスト環境の将来の展望として、テスト環境に実装している割り当て方法をすべて実行し、その結果を 5 章で提案した測定方法に基づき、最も適した割り当て方法を自動生成する。これは近い将来、実装が可能である。

また、実機を用いたテスト環境の将来の展望として、割り当て方法及びパラメータを自動的に導出するテスト環境がある。各アプリケーションをテスト実行することにより、自動的に割り当て方法や、パラメータを導出できれば理想的である。しかし、入力を待つようなアプリケーションを対象とする場合は自動的に割り当て方法を導くことは困難である。

しかし、対象とする機器に連動したテスト環境の構築し、動作環境ごとの割り当て方法、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自動的に割り当て方法やパラメータの導出の足掛かりとなると期待できる。

もし、実行環境やアプリケーションによって、割り当て方法やパラメータを自動的に導出することができれば、プログラマは今まで以上にメモリ管理を気にすることなくアプリケーションの作成が可能になると期待している。

謝辞

本研究を行うにあたり、有益な御指導、助言および授助を頂きました権藤克彦助教授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教授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に深く感謝致します。

関連図書

[1] Paul R. Wilson, Mark S. Johnstone, Michael Neely, and David Boles, Dynamic Storage Allocation: A Survey and Critical Review In Proceedings of the international Workshop on Memory Management, 1995

[2] Mark S. Johnstone, Paul R. Wilson, The Memory Fragmentation Problem: solved?, In Proceedings of the International Workshop on Memory management pp26-36 ACM, 1998

[3] Kenneth C. Knowlton, A Fast Storage Allocator, Communicatons of the ACM, 8(10):623-625, October 1965

[4] K.K.Shen and J.L.Peterson, A weighted buddy method for dynamic storage allocation, Communicatons of the ACM, 17(10):558-562, October 1974

[5] D.E.Knuth, 米田信夫, 筧捷彦 共訳, 2 基本算法/情報構造, サイエンス社, 1978

[6] T.G. レヴィス, M.Z. スミス 著, 浦昭二, 近藤頌子, 遠山元道 共訳, 情報処理シリーズ 10　データ構造, 培風館, ISBN 4-563-00790-0, 1987

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

[8] Daniel P.Bovet, Marco Cesati 著, 高橋浩和, 早川仁 監訳, 岡島順治郎, 田宮まや, 三浦広志 訳, 詳細　LINUX カーネル, オライリージャパン, ISBN 4-87311-048-3, 2001

[9] Bonwick, j., The Slab Allocator: An Objected-Caching Kernel Memory Allocator, Proceedings of the Summer 1994 USENIX Technical Conference, Jun. 1994, pp87-98

[10] Uresh Vahalia 著, 徳田秀幸, 中村明, 戸辺義人, 津田悦幸訳, 最前線 UNIX のカーネル, ピアソン・エデュケーション, ISBN4-89471-189-3, 2000

[11] 日比野靖, ごみ集めの基本アルゴリズム, 情報処理学会誌 vol.35 No.11 p992-999, 1994

[12] Neil Rbodes, Julie McKeeban 共著, 青柳龍也 監訳, 佐藤信彦 訳, Palm プログラミング, オライリージャパン, ISBN 4-87311-009-2, 1999

[13] 内山雄司, 脇田健, メモリ管理機能のモジュラーかつ効率的な実装手法, 情報処理学会論文誌：プログラミングに掲載予定

[14] Drerek White, Alex Garthwaite, The GC Interface in the EVM, Sun Microsystems http://research.sun.com/techrep/1998/abstract-67.html, 1998

[15] Giuseppe Attardi, Tito Flagella, A Customisable Memory Management Framework, Technical Report TR-94-010, International Computr Science Institute, Berkeley, 1994

付録：ガベージコレクション[7][11]

ここでは、直接的に本研究には関係ないが、メモリ管理に非常に重要なガベージコレクション (GC) について、簡単に用語説明と GC の方法を説明する。

ガベージ (ゴミ) とは

ガベージとは、再び使用することのないヒープ上のメモリ領域のことである。ルートからポインタを介してアクセスできない領域はガベージと判断され、空きリストに返される。

ガベージコレクションとは

ガベージコレクションは、もはやどこからも参照されないようなオブジェクトを自動的に解放するための技術で、解放のし忘れ (メモリリーク) や、誤ったオブジェクトの解放によって発生するダングリング参照の問題を自動的に対処することができる。このため、明示的なメモリ管理に比べ、保守性が向上する。

基本的な GC アルゴリズム

ここでは、基本的な GC アルゴリズム (reference counting、mark-sweep、opying 法) の説明を行う。

Reference counting 法

reference counting 法は、アクティブセルやルートから各セルへの参照数を数えることで、セルが使われているかどうかの判別をする。このアルゴリズムの利点は、いつでも参照数の変化を認識しているため、GC 処理を分散できることである。

しかし、以下のように欠点もいくつかある。

- 単純なアルゴリズムではユーザプログラム中に処理が分散するので、処理が重くなる。

- 循環データ構造の回収ができないため、回収しようとすれば他の GC アルゴリズムと組合合わせなければ解決できない。
- 参照数を保持するために余分な領域が必要となる。参照数は正確でなければならないので、全体のポインタの数を示せるだけの領域が必要となる。

Mark-sweep 法

mark-sweep 法は、ガベージが生成されるとすぐに回収を行う reference counting 法とは異なり、利用可能な領域が使い尽くされてからはじめて GC を起動する。このため、GC 起動中にはユーザプログラムは停止する。GC によって十分メモリを回収できれば、ユーザプログラムは再開される。ガベージかどうかの判別は、ルートからたどって到達可能でないオブジェクトをガベージと判断する。

mark-sweep 法は mark フェーズと sweep フェーズの 2 段階で動作する。まず、mark フェーズでルートからたどることのできるセルを識別する。このとき、到達可能かどうかの情報を保存するため、mark-bit を各セルに余分に用意しなければならない。mark フェーズが終了すると、sweep フェーズでヒープ全体を走査し、mark-bit の情報をもとに到達可能でないセルを空きリストに返す。

mark-sweep 法は reference counting 法に勝る点が 2 つある。1 つはポインタ操作にオーバーヘッドがないこと。もう 1 つは循環データ構造が回収できることである。しかし GC 中はユーザプログラムが停止するため、リアルタイムシステム、会話型システムやビデオゲームには適さない場合もある。また、sweep フェーズでヒープ全体を走査するので計算量はヒープの大きさに比例する。

Copying 法

copying 法はヒープを均等に 2 分割し、一方の利用可能な領域が使い尽くされると GC を起動し、到達可能なセルをもう一方の領域にコピーするという方針をとっている。このアルゴリズムの利点は、コピーするときにセルを詰めなおすので断片化の発生を心配しなくてよいことである。

copying 法は mark-swee 法や reference counting 法と比べると広く利用されている方法である。コンパクションをおこなわない方法と比べると、可変サイズのセルが要求される場合の割り当てコストを軽減できる。しかし、ヒープ全体の半分の領域しか利用できないという欠点をもつ。

Mark-sweep 法と Copying 法の比較

ここでは、mark-sweep 法と copying 法の比較をする。しかし、仮想メモリやキャッシュの影響や、割り当てコストについて考えないものとする。

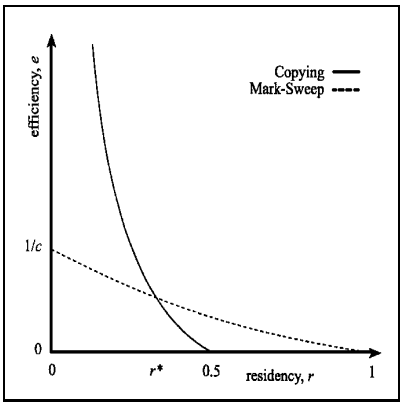

図 7.1: 効率とプログラムとメモリの占有率の関係

ヒープの大きさを M、ライブメモリの総量を R とする。ここで、copying コレクタの時間複雑度 t_{Copy} を考える。copying コレクタは、ルート集合とアクティブデータグラフ中のすべてのポインタを走査し、アップデートをおこない、Tospace のオブジェクトを回収する。従って、時間複雑度は R に近づく。

$$t_{Copy} = a\mathrm{R}$$

次に、mark-sweep コレクタの時間複雑度 t_{MS} について考える。mark-sweep コレクタは、マークフェーズでライブデータ構造を追跡し、スィープフェーズでヒープ全体を線形探索する。従って、時間複雑度は以下の式で示される。

$$t_{MS} = b\mathrm{R} + c\mathrm{M}$$

GC によって回収されるメモリの総量はそれぞれ以下の式で示される。

$$m_{Copy} = \frac{\mathrm{M}}{2} - \mathrm{R}$$
$$m_{MS} = \mathrm{M} - \mathrm{R}$$

単位時間あたりの回収するメモリの総量、つまりアルゴリズムの効率を e と定義すると、それぞれの効率は以下の式で示される。

$$e_{Copy} = \frac{1}{2ar} - \frac{1}{a}$$
$$e_{MS} = \frac{1-r}{br+c} \qquad r = \frac{\mathrm{R}}{\mathrm{M}}(\text{プログラムの占有率})$$

図 7.1 を見ると、メモリの占有率が小さい (ヒープ領域が十分に大きい) 場合は、mark-sweep 法よりも copying 法が効率が良い。しかし、ある占有率 $r*$ よりも占有率が大きくなると、mark-sweep 法の効率が勝る。

ここで使用した引数の大小関係は、$a > b > c$ となる。なぜなら、オブジェクのコピーは、mark-bit の設定よりもコストが高いと予測できる $(a > c)$。また、ヒープの線形探索よりも、データ構造の追跡がコストが高いと予測できるからである $(b > c)$。

しかし、ここで説明した GC の基本アルゴリズムは非常に単純で、改善の余地を残しており、実際はさらに複雑なアルゴリズムとなる。

copying 法が、mark-sweep 法よりも優れていると言われてきた。しかし、最近の研究では mark-sweep 法か copying 法かの選択は、クライアントプログラムの動作と GC アルゴ

リズム固有の特性によって異なるといわれている。 本研究では、組み込み機器を対象としているので、mark-sweep 法が適切であると主張する。

Consrvative GC

これまで、GC は常にあらゆるメモリ内容の型を知ることができるということを前提としていた。しかし、C 言語のように、実行時に型がわからない環境での GC をおこなう処理系も存在し、Boehm GC などが有名である。

それでは、ポインタか否かの判断をどうするのかといういと、まずはポインタであると仮定するという方針をとる。そして、ヒープ領域外を指していたり、空き領域であるはずの個所を指している場合には、明らかにポインタでないと判断することができる。

しかし、この方針をとる限り、ユーザプログラムがポインタのつもりで使っていない値を GC がポインタだと判断する可能性は避けられない。

Generational GC

セルの寿命について多くのプログラムで観測されるある傾向が知られている。それは、ほとんどのオブジェクトは生成後、間もなく死んでしまい、ある程度長く生き残ったオブジェクトは長期的に生きつづけるという傾向である。

generational GC は copying 法をベースにする場合は、寿命の長さを判定し、2 つ以上世代 (generational) に分類して異なる領域に割り当てる。GC で生き残ったセルは、1 つ上の世代へ移行するためのカウントが増やされる。カウントが一定の数を超えると、上の世代に昇進する。また、通常の GC は最も若い世代のみを GC の対象とし、空き領域が不足すると、はじめて 1 つ上の世代を GC する。さらに空き領域が不足した場合は、さらに上の世代を加えて GC をする。この結果、GC の頻度を減少することができる。

しかし、世代間をまたがって参照がある場合には、誤って使用中のオブジェクトを回収してしまうという問題点がある。このため、若いオブジェクトが古い世代のオブジェクトを参照している場合は、若いオブジェクトを特別扱いしなければならない。メモリオブジェクトへの書き込みの際に特別な処理を行う機構を write barrier という。

また、generational GC が有効なのは、最初に述べた傾向に良く当てはまっている場合である。この場合、一度の GC 時間が短く、解放できるメモリ量は他の GC と変わらないため、全体の実行時間、プログラムの停止時間の両方を改善できる。しかし、オブジェクトの生存率が高い場合には、この方法で停止時間を抑えることは難しい。

Incremental GC

Generational GC は、単純な GC に比べれば、プログラムの停止時間を短くすることができる。しかし、それでもリアルタイム性が必要なプログラムにとっては長すぎる場合が

ある。incremental GC は GC 処理を分割し、ユーザプログラムと交互に動作させることによってを GC による 1 回ポーズ時間を短縮することを可能にする割り当て方法である。

基本的な問題点は、ユーザプログラムがヒープオブジェクトの書き換えをすることで、GC が誤って使用中のオブジェクトを回収する可能性があることである。このため、ユーザプログラムによる変更を GC が知る必要がある。

しかし、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なく、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。このため、記憶領域の乏しい組み込み機器には必ずしも適さない。