プログラミング 16-6
(1997. 11. 21)

オブジェクトの世代を考慮に入れた保守的ごみ集め

萩原 知章　岩井 輝男　中西正和

慶應義塾大学大学院 理工学研究科 計算機科学専攻

C 言語では，自動メモリ管理システムがライブラリの形で実装されている．実行時に型に関する情報を得られない環境で，保守的に行なうことでごみ回収 (以下 GC) を実現している．しかし，通常の malloc / free で管理する方法に比べ，メモリ使用量が増えてしまう．この原因の 1 つとして考えられるのが生成後すぐに不必要になるオブジェクトが即時に回収されないことである．本稿ではこの点を改善するため自動メモリ管理のごみ回収部分に世代別の考え方を導入した GC を提案し，この有効性を示す．

Conservative Garbage Collection Considering the Objects' life time

Tomoaki HAGIWARA　Teruo IWAI　Masakazu NAKANISHI

Department of Computer Science
Graduate School of Science and Technology
Keio University
3-14-1, Hiyosi, Kouhoku, Yokohama 223, Japan

The automatic memory management system is implemented for C as a library. This system does not expect pointers to be tagged, so collector finds the dead objects conservatively. But the conservative garbage collection uses more heap space than the explicit memory management. One of this reason is that collector doesn't collect immediately the dead objects not living long. This paper shows that the generation collection is effective for the conservative garbage collection.

1 はじめに

不必要になったオブジェクトに割り当てられていたメモリを再利用する機能 (以下 GC) が実装されている言語では，通常オブジェクトの型に関する情報を実行時に得ることが可能である．

それに対して C 言語では実行時に型に関する情報が得られない．こうした環境の中で，Boehm は，ライブラリの形で C 言語に自動メモリ管理を実装した [2][1].

本稿では自動メモリ管理のごみ回収部分に世代別の考え方を採り入れた GC を実装する．世代別 GC では 1 回の GC にかかる時間を短縮できるため，より頻繁に GC を行なうことが可能となる．これにより，メモリ使用量を減らすことが可能となる．

2 Boehm の自動メモリ管理

Boehm の実装したヒープの自動メモリ管理について述べる [3][5].

2.1 ヒープ管理

自動メモリ管理を用いているプログラムでは 2 つの論理的に区別されたヒープが共存している．1 つは「自動メモリ管理を行なうヒープ」であり，もう 1 つは「通常の malloc / free システムコールで管理を行なうヒープ」である．自動メモリ管理を行なうヒープから malloc / free で管理を行なうヒープにあるオブジェクトを指すことあるが，このオブジェクトを GC により回収することはない．したがって，C 言語のライブラリの中で，malloc / free でメモリを管理するコードが含まれている場合でも，自動メモリ管理を用いることが可能である．

2.2 メモリ割り当て方法

自動メモリ管理を行なうヒープ領域はいくつかの *block* から成り立っている．この *block* は 4 Kbytes である (多くの UNIX で採用されている ページの大きさと同じになっている)．ヒープ領域を拡張する際には，オペレーティング・システムからいくつか *block* を獲得する．そして，図 1 のようにこの獲得された塊にヘッダを付け，*block* 用のフリーリストに繋ぐ．メモリ獲得の方法は要求されている大きさが 2 Kbytes より大きい時と 2 Kbytes 以下の時とでは異なっている．

2K bytes よりも大きい場合

基本的に first fit 戦略を使って *block* 用のフリーリストから必要な大きさの *block* を取ってくる．要求された大きさが 4K bytes よりも大きな場合はいつくかの連続した *block* を用いる．十分な大きさの連続した *block* がなければ，GC を起動する．それでも必要大きさの領域が見つからない時にはヒープ領域を拡張する．

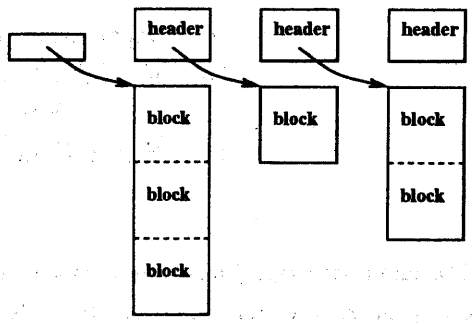

図 1: *block* 用のフリーリスト

2 Kbytes 以下の場合

割り当てることの可能なメモリの大きさはあらかじめ数種類に限定している (4, 8, 16, 32, 48 ... 2K bytes). そして図 2 のように各大きさごとにフリーリストが設けてある.

メモリ要求がある場合, その大きさから実際に割り当てる大きさを計算し, この大きさのフリーリストにある先頭のものを割り当てる (たとえば 14bytes のメモリ要求があった場合実際には 16bytes の大きさが割り当てられる).

ある大きさ (n bytes) のフリーリストが空のときには, GC を起動する. それでも n bytes 用のフリーリストが空ならば, 前述の *block* 用のフリーリストから 1 *block* 領域を確保する. これにヘッダを付け, n bytes ごとにくぎり, フリーリストに繋げる.

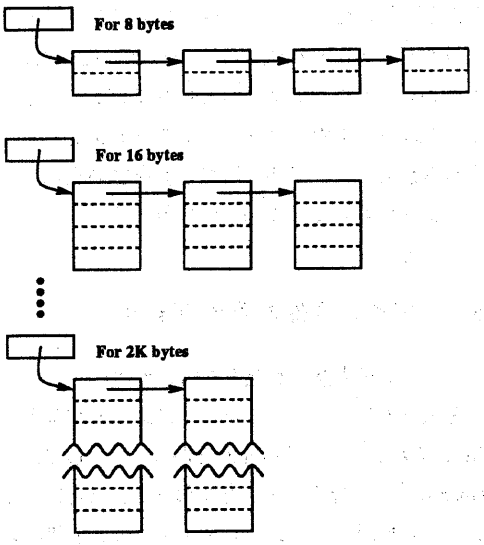

図 2: 2K bytes 以下のメモリのフリーリスト

2.3 ごみ回収

C 言語ではデータの型に関する情報を得ることが不可能である. そのため回収器はポインタである可能性のある word を探しだし, そのアドレスが指すオブジェクトを生きているものとみなす. このようなポインタを潜在的ポインタ (**potaintial pointer**) と呼ぶ.

回収アルゴリズム

潜在的ポインタは実際にはポインタでない場合がある. そのため, この潜在的ポインタを書き換える必要のあるコピー法を用いることはできない. したがって, GC の回収アルゴリズムとして, マークスイープ法を用いる (印づけする領域はヘッダに取られている).

潜在的ポインタであるかの判断方法

潜在的ポインタであるかの判断は正確にかつ簡単に決定できることが必要である. これを判断するテストは 3 つある. p は次のテストをクリアした時に潜在的ポインタとみなされる.

1. p が指しているオブジェクトは自動メモリ管理を行なうヒープに属しているか
2. p が指しているオブジェクトは既に割り当てられたものか (フリーリストに繋がれていないか)
3. p はオブジェクトの先頭を指しているか

印づけ

まず, レジスタ, 静的領域, スタックの中からルートとなる潜在的ポインタを探しだす.

そしてルートから順に印づけを行なっていき, 印づけされたものは GC 用のスタックに乗せられる.

そしてこの GC 用スタックが空になるまで，印づけが行なわれる．

印づけが終った時点で印づけがされていないオブジェクトはごみである．ごみとなった領域が 2Kbytes より大きいものは *block* 用のフリーリストに繋ぎ，2Kbytes 以下のものはその大きさに対するフリーリストに繋げる．

3 性能比較

Zorn は Boehm の自動メモリ管理と malloc / free で管理を行なう場合についての性能比較を行なった．両者の実行時間は大差がないが，自動メモリ管理を用いた場合，図 3, 4 をみればわかるように最大メモリ使用量が 1.2 倍〜 2.4 倍に増えてしまうという報告されている [4].

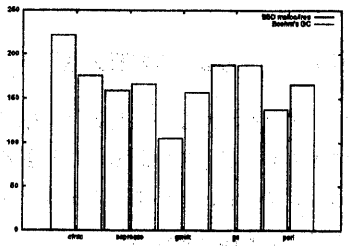

図 3: 実行時間

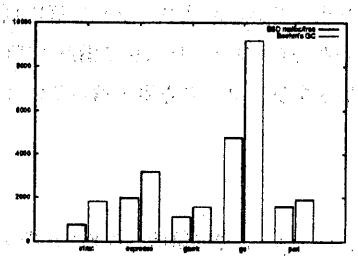

図 4: 最大メモリ使用量

4 本研究の方針

「ほとんどのオブジェクトは生成後すぐに死んでしまい，ある程度生き続けたオブジェクトは半永久的に生き続ける」という報告がされている．これは従来 Lisp において言われてきたことだが，C 言語でも同様な報告がされている．

このことを考えれば，自動メモリ管理を用いた場合，生成後すぐにごみとなるオブジェクトが長時間回収されないことが最大メモリ使用量を増やしてしまう原因の 1 つとななることが予想できる．

本稿では自動メモリ管理のごみ回収部分に世代別の考え方を導入した GC を提案し，この有効性を示す．若い世代のオブジェクトに対し，GC を頻繁に行なうことにより，生成後すぐにごみとなるオブジェクトをより早く回収し，最大メモリ使用量を減らす．

ごみ回収の際には **full cycle** と **partial cycle** の 2 種類のサイクルを用いる．full cycle では通常の GC と同様に生きているすべてのオブジェクトに印づけを行なう．それに対し，partial cycle では古い世代のオブジェクトはすべて生きているものとみなし，生成されてから間もないオブジェクトとポインタ書き換えによって印づけが必要な古い世代のオブジェクトに対してのみ印づけを行なう．

4.1 ポインタ書き換えの検出

ポインタ書き換えが起きた古い世代のオブジェクトを検出するために，このオブジェクトが含まれる *block* に対し，メモリの保護をかける．そして，この *block* に対し，書き込みを行なおうとした時に，このアドレスを調べる．このアドレスに古い世代のオブジェクトが存在し，かつ書き込みを行なおうとするものが，潜在的ポインタならば，これをヘッダに保存する．このポインタは partial cycle の際にルート

となる．

このアドレスを調べるにはオペレーティング・システムの発するシグナルを受けとることによって行なっている．そのため 1 つの *block* に頻繁に書き込みが行なわれると実行速度が低下する．そこで，保護をかけた *block* に一定回数以上書き込みがあった場合には保護を外す．そして partial cycle の際にこの *block* に含まれるすべての古い世代のオブジェクトをルートとする．

5 実験および結果

本稿の実験では，世代別 GC と従来型の GC について，ベンチマークプログラムを用いて実行時間 (ユーザ時間，システム時間)，最大メモリ使用量，GC の起動回数を計測した．実験環境を以下に示す．ベンチマークプログラムには espresso,ghostscript を用いた．この実験結果を表 1, 表 2 に示す．

実験環境

- 機種 — SPARC station 20
- 主記憶 — 320MB
- オペレーティングシステム — SunOS 5.5.1

表 1: espresso

	世代別	従来型
ユーザ時間 (秒)	28.2	31.0
システム時間 (秒)	4.1	0.6
実行時間 (秒)	32.3	31.6
最大メモリ使用量 (Kbytes)	1519	1971
GC 起動回数	293	273

表 2: ghostview

	世代別	従来型
ユーザ時間 (秒)	38.6	38.8
システム時間 (秒)	6.6	1.8
実行時間 (秒)	45.2	40.6
最大メモリ使用量 (Kbytes)	22,676	20,255
GC 起動回数	42	46

図 1,2 を見ればわかるように，世代別 GC では従来型の GC に比べ，最大メモリ使用量が少なくなっている (espresso では 23% ghostscript では 12% 減少している)．これは従来型に比べ GC の回数を多くしているためであると考えられる．また，partial cycle を用いているため，1 回あたりの GC の時間が減少している．そのためユーザ時間は減少している．

しかし，システム時間が従来型に比べ多くかかっている．これはポインタ書き換えが起きた古い世代のオブジェクトを検出するためにオペレーティング・システムの発するシグナルを受けとることに時間がかかっていると思われる．

6 結果

C 言語のごみ回収に世代別の考え方を採り入れた場合では従来型に比べ，メモリ使用量を抑えることができることが分かった．しかし，システム時間が多くかかるため，全体の実行時間が多くかかってしまう．この点については改善の余地があると考えられる．

7 今後の展望

ルートとなる潜在的ポインタが含まれる領域はレジスタ，静的領域，スタックである．このうち実行時に最も領域の大きさが変化するのはスタック領域

である．スタック領域が小さい時にごみ回収を行なうことにより，ごみ回収の実行速度を上げることが可能であると考えられる．

参考文献

[1] Hans-Juergen Boehm. Space efficient conservative garbage collection. In *Proceedings of SIGPLAN'93 Conference on Programming Languages Design and Implementation*, Vol. 28(6) of *ACM SIGPLAN Notices*, pp. 197–206. ACM Press, June 1993.

[2] Hans-Juergen Boehm and Mark Weiser. Garbage collection in an uncooperative environment. *Software Practice and Experience*, Vol. 18, No. 9, pp. 807–820, 1988.

[3] R. Jones and R. Lins. *Garbage Collection – Algorithms for Automatic Dynamic Memory Management*. John Wiley and Sons Ltd, 1996.

[4] Benjamin Zorn. The measured cost of conservative garbage collection. Technical Report CU-CS-573-92, 1992.

[5] 小野寺民也. 保守的ごみ集め. 情報処理, Vol. 35, No. 11, pp. 1020–1026, 1994.