Mark-Sweep Garbage Collection

Hagen Böhm
(hagen@net.uni-sb.de)

Seminar "Garbage Collection" ,
Universität des Saarlandes

WS 2001 (Version 24. Juni 2002)

1 Einleitung

Seit Beginn der 1960er Jahre wurden mehrere verschiedene Ansätze zur automatischen Wiedergewinnung von Speicher entwickelt. Eine davon ist *tracing garbage collection*, deren ältester Vertreter, der Mark-Sweep Garbage Collector auch gleichzeitig der älteste Algorithmus seiner Art ist. Dieser Artikel beschäftigt sich mit der Arbeitsweise von Mark-Sweep und diskutiert Lösungsansätze zu bestimmten Schwachstellen.
Zunächst wird in Kapitel 2 die Funktionalität des Algorithmus in ursprünglicher Form vorgestellt. Daran anschließend diskutieren wir einen iterativen Markierungs-Algorithmus und die dabei auftretende Problematik des Keller-Überlaufs. In Kapitel 4 wird eine Technik zum Markieren mit konstantem Speicher vorgestellt und Kapitel 5 diskutiert die "Auslagerung" der Markierungs-Bits in eine Bitmap Tabelle. Schließlich beschäftigen wir uns noch mit dem sogenannten *lazy sweeping*, einer Technik, die das Aufräumen des Speichers in Stücken auf die Allokation überträgt.
Abschließend gibt Kapitel 7 einen Überblick über Vor- und Nachteile von Mark-Sweep.
Im folgenden werden noch einige Programm-Teile, die im Text verwendet werden näher erläutert:

`Roots`	Menge der Wurzel-Knoten einer Speicher-Halde
`free_pool`	Menge der frei zur Verfügung stehenden Speicher-Zellen
`Children(X)`	Menge der Verweise eines Knotens X (Objekt)
`*Y`	Wert eines Verweises Y nachdem er dereferenziert wurde
`free(X)`	Methode, die als Parameter einen Knoten X nimmt und diesen dem `free_pool` hinzufügt
`Heap`	Speicher-Halde

2 Der Mark-Sweep Algorithmus

Die automatische Wiedergewinnung von Speicher durch Mark-Sweep läuft in ihrer ursprünglichen Form wie folgt ab:
Sobald ein laufender Prozeß eine Speicheranforderung stellt, der Bedarf an Speicher jedoch nicht oder nur zum Teil befriedigt werden kann, wird der Prozeß angehalten und Mark-Sweep gestarted. Der Ablauf von Mark-Sweep teilt sich in zwei Phasen auf: Phase 1, die *Markierungs-Phase* (Mark-Phase), traversiert die Programm-Halde beginnend bei den Wurzeln, indem sie den Verweisen folgt und markiert dabei alle Objekte, die sie erreichen kann und somit erreichbar sind. Demnach sind die Objekte, die am Ende der Phase unmarkiert verbleiben unbenutzt und können freigegeben werden.
Nun beginnt Phase 2, die *Aufräum-Phase* (Sweep-Phase). Sie traversiert den kompletten Speicher linear von der ersten bis zur letzten Zelle. Dabei gibt sie alle nicht markierten Objekte frei und löscht die Markierungen der übrigen. Wenn nach Ablauf von Phase 2 ausreichend Speicher freigegeben wurde, wird der anfordernde Prozeß

wieder aufgenommen, andernfalls muß entweder die Halde erweitert, oder der Prozeß abgebrochen werden.
Wegen der Vorgehensweise der beiden Phasen (das Traversieren von Speicher) wird Mark-Sweep als *tracing garbage collection* bezeichnet. Abbildung 1 zeigt eine abstrahierte Form von Mark-Sweep in Pseudo-Code.

```
mark_sweep() =
    for R in Roots
        mark(R)
    sweep()
    if free_pool is empty
        abort ''Memory exhausted''
```

Abbildung 1: Simpler Mark-Sweep garbage collector

Die Markierungs-Phase, also das Traversieren und gleichzeitige Markieren der Programm-Halde kann man durch einen einfachen, rekursiven Ansatz verwirklichen, wie er in Abbildung 2 dargestellt ist. Hierbei werden mittels Tiefen-Suche die einzelnen Objekte der Halde rekursiv inspiziert und markiert. Die Rekursion terminiert in jedem Fall, da die Halde endlich ist und Nachfolger von bereits markierten Knoten (Objekten) nicht mehr inspiziert werden.

```
mark(N) =
    if mark_bit(N) == unmarked
        mark_bit(N) = marked
        for M in Children(N)
            mark(*M)
```

Abbildung 2: Markierung durch einfache Rekursion

In der, an die Markierungs-Phase anschließenden Aufräum-Phase müssen die nicht mehr benutzten (nicht markierten) Objekte dem Betriebssystem wieder zur Verfügung gestellt werden und die Markierungen für einen erneuten Durchlauf von Mark-Sweep rückgängig gemacht werden. Abbildung 3 zeigt einen Algorithmus, der diese Aufgabe durch globales Traversieren des Speichers löst. Mark-Sweep besitzt gegenüber *Reference Counting* zwei bedeutende Vorteile. Zum einen können zyklische Programmstrukturen ohne zusätzlichen Aufwand behandelt werden, zum anderen gibt es keine Verwaltung von Verweisen, die sich bei der Methode Reference Counting in ihrer Handhabung als sehr aufwendig erwiesen hat. Nichtsdestotrotz besitzt auch Mark-Sweep Schwächen. Im folgenden wollen wir uns daher mit der Optimierung einiger dieser Schwächen beschäftigen.

```
sweep() =
    N = Heap_bottom
    while N < Heap_top
        if mark_bit(N) == unmarked
            free(N)
        else mark_bit(N) = unmarked
        N = N + size(N)
```

Abbildung 3: Aufräumen durch globalen Speicherdurchlauf

3 Iterativ statt rekursiv markieren

Rekursive Prozedur Aufrufe wie sie in Kapitel 2 verwendet werden um die Markierung der Halde durchzuführen, sind auch heute noch relativ Zeit- und Speicherintensiv und können ab einer gewissen Tiefe der Programm Halde zum Überlauf des Systems führen.
Wenn man stattdessen einen Hilfs-Keller anlegt und die rekursiven Aufrufe durch iterative Schleifen ersetzt, kann man den Aufwand für die Rekursion umgehen. In einem solchen Keller werden Verweise auf Knoten abgelegt, die erreichbar aber noch nicht besucht worden sind, und in iterativen Schleifen sequentiell abgearbeitet. Dies ist wesentlich effizienter und auch einfacher zu handhaben. Abbildung 4 zeigt zwei Ansätze zur Simulation der Rekursion durch Iteration mit Hilfe eines Kellers. Die maximale Tiefe des Kellers hängt dabei stark von der Struktur der Programm Halde ab.

```
mark_heap() =
    mark_stack = empty
    for R in Roots
        mark_bit(R) = marked
        push(R, mark_stack)
        mark()

mark() =
    while mark_stack != empty
        N = pop(mark_stack)
        for M in Children(N)
            if mark_bit(*M) == unmarked
                mark_bit(*M) = marked
                if not atom(*M)
                    push(*M, mark_stack)
```

a) Traversiert jeden Knoten

```
mark_heap() =
    mark_stack = empty
    for R in Roots
        push(R, marck_stack)
        mark()

mark() =
    while mark_stack != empty
        N = pop(mark_stack)
        if mark_bit(N) == unmarked
            mark_bit(N) = marked
            for M in Children(N)
                push(*M, mark_stack)
```

b) Traversiert jeden Verweis

Abbildung 4: Iterative Markierungs-Algorithmen

Der Unterschied zwischen den beiden iterativen Algorithmen besteht darin, daß Algorithmus a) jeden Knoten **nur einmal** auf den Keller legt (ausgenommen Blatt-Knoten), während Algorithmus b) jeden Knoten **so oft inspiziert, wie Verwei-**

se ihn referenzieren. Algorithmus a) ist somit im Vorteil gegenüber b) falls der Programm-Graph mehrfach referenzierte Knoten besitzt. Allerdings bringen sowohl a) als auch b) den Hilfs-Keller zum Überlaufen, wenn sie Knoten erreichen, die mehr Kinder-Knoten haben als freie Keller-Plätze zur Verfügung stehen (z.B. große Arrays). Um die Häufigkeit des Auftretens dieses Problems zu verringern, werden die Kinder-Knoten nicht vollständig sondern in kleinen Gruppen auf den Keller gelegt. Eine Gruppe kommt erst dann auf den Keller, wenn der von der vorherigen Gruppe aufgespannte Teil-Graph vollständig abgearbeitet wurde. Dieses Verfahren kann durch zusätzliche Hilfs-Zeiger realisiert werden und ist in [Boehm and Weiser, 1988] beschrieben.

3.1 Keller Überlauf

Trotz der im vorherigen Abschnitt beschriebenen Präventivmaßnahme kann es natürlich immer noch zu Keller Überläufen kommen. Wenn dies jedoch auf einem Hilfs-Keller geschieht, läßt sich darauf wesentlich besser reagieren, als es bei einem Überlauf bedingt durch Rekursion der Fall ist. Im folgenden wollen wir uns mit der Erkennung und Behandlung von Keller Überläufen beschäftigen.

3.1.1 Erkennung eines Keller Überlaufs

Im wesentlichen gibt es zwei Möglichkeiten einen Überlauf zu entdecken. Die erste ist simpel: Man erweitert die *push*-Operation um einen Speicher-Check, so daß push jedesmal wenn es einen Knoten auf den Keller legt überprüft, ob noch genügend Speicher vorhanden ist.
Eine effizientere Methode ist die Überprüfung des Keller-Speichers vor Beginn eines Durchlaufs der Markierungs-Schleife. Sobald ein Knoten vom Keller geholt wurde wird die Anzahl seiner Kinder mit der Anzahl der verfügbaren Speicher-Plätze verglichen. Ist genügend Platz vorhanden, wird die Schleife gestartet, andernfalls ein Keller Überlauf gemeldet.

3.1.2 Behandlung von Keller Überläufen

Auch hier gibt es wieder mehrere Möglichkeiten, wenn ein Überlauf entdeckt wurde. **Knuth** [Knuth 1973] schlägt vor, einen zirkulären Keller zu benutzen. Bei einer Keller Größe von n Plätzen wird der i-te Knoten an Stelle $i \bmod n$ eingefügt. Daraus folgt, daß wenn mehr als n Knoten auf den Keller gelegt werden, einige davon überschrieben werden müssen und somit Wurzel-Knoten verloren gehen, deren zugehörige Teil-Graphen dann nicht mehr markiert werden können.
Zur Lösung dieses Problems schlägt Knuth vor, die Programm-Halde nach markierten Knoten zu scannen, die unmarkierte Kinder haben. Ein solcher Scan setzt immer dann ein, wenn der Keller leergelaufen ist und zuvor Knoten überschrieben wurden. Die so gefundenen Kinder Knoten dienen als Wurzeln des Teil-Graphen an dem nun die Markierungs-Phase wieder aufgenommen wird. Der Prozeß terminiert schließlich, wenn der Keller leergelaufen ist, ohne daß zuvor Einträge überschrieben wurden.

Zur Steigerung der Effizienz läßt man den Scan am überschriebenen Knoten mit der niedrigsten Adresse, statt am Anfang der Programm-Halde beginnen.
Eine Variante der Lösung von Knuth stellt der gemeinhin als **Boehm-Demers-Weiser** collector bekannte garbage collection Algorithmus dar, auf den ich in Kapitel 6 noch kurz eingehen werde. Dieser unterscheidet sich von Knuth wie folgt: Statt alte Einträge im Keller mit neuen zu überschreiben, wenn der Keller vollständig gefüllt ist, werden letztere einfach verworfen. Trat ein Überlauf auf, so wird außerdem die Größe des Kellers verdoppelt, nachdem er leergelaufen ist.
Obwohl nicht bekannt ist, welche Effekte die beiden Strategien haben, so wird vermutet [Jones and Lins, 1996], daß Boehm-Demers-Weiser ein besseres Cache-Verhalten aufweist.

4 Markieren mit konstantem Speicher

Auch wenn es Methoden zur Erkennung und Behebung von Keller Überläufen gibt, so stellt sich dennoch die Frage nach ihrer Vermeidung. Da effizientes Markieren von Halde-Graphen nicht ohne Zwischenspeichern der traversierten Verzweigungspunkte möglich ist, muß der Keller jedoch auf irgendeine Art erhalten bleiben. Als Lösung bietet sich an, den Graphen selbst als Unterbringungsort für die Informationen aus dem Keller zu nutzen. Für das Markieren des Programm-Graphen wenden wir dazu eine Technik namens **pointer-reversal** an, die unabhängig voneinander von [Schorr and Waite, 1967] und Deutsch [Knuth, 1973, Exercise 2.3.5.8] entwickelt wurde. Hierbei handelt es sich um eine Methode, bei der die Verweise der Knoten kurzfristig umgedreht, d.h. vom Kind auf den Vater des Knotens umgebogen werden. Dies geschieht, um zuvor traversierte Verzweigungspunkte vom aktuellen Standort des Markierens aus erreichbar zu halten.
Im folgenden Abschnitt wird diese Technik zunächst an Graphen vorgestellt, deren Verzweigungsknoten genau zwei Verweise enthalten, nämlich auf ein linkes und auf ein rechtes Kind. Anschließend wird erklärt wie sie auf Knoten mit beliebig vielen Verweisen erweitert werden kann.

4.1 *Pointer reversal* auf binären Graphen

Der Deutsch-Schorr-Waite Pointer Reversal Algorithmus zum Markieren des Halde-Graphen läßt sich durch einen endlichen Automaten mit drei Zuständen beschreiben. Eine schematische Darstellung ist in Abbildung 5 zu sehen. Jeder der Zustände steht dabei für eine Phase des Algorithmus. In Phase 1 werden so lange linke Kinder traversiert, bis ein Blatt-Knoten oder ein bereits markierter Knoten erreicht wurde. Dabei wird jeder besuchte Knoten markiert und der Verweis auf sein linkes Kind mit einem Verweis auf seinen Vater-Knoten überschrieben. Auf diese Weise bleiben sämtliche, bereits durchlaufene Verzweigungspunkte erreichbar.
Phase 1 stoppt, wenn der Automat aus dem Zustand *gehe zu linkem Kind* in den Zustand *zurückziehen* wechselt. In diesem Zustand beginnt Phase 2, in der wir uns über die konstruierten Verweise auf die Vater-Knoten (pointer reversal) zu dem

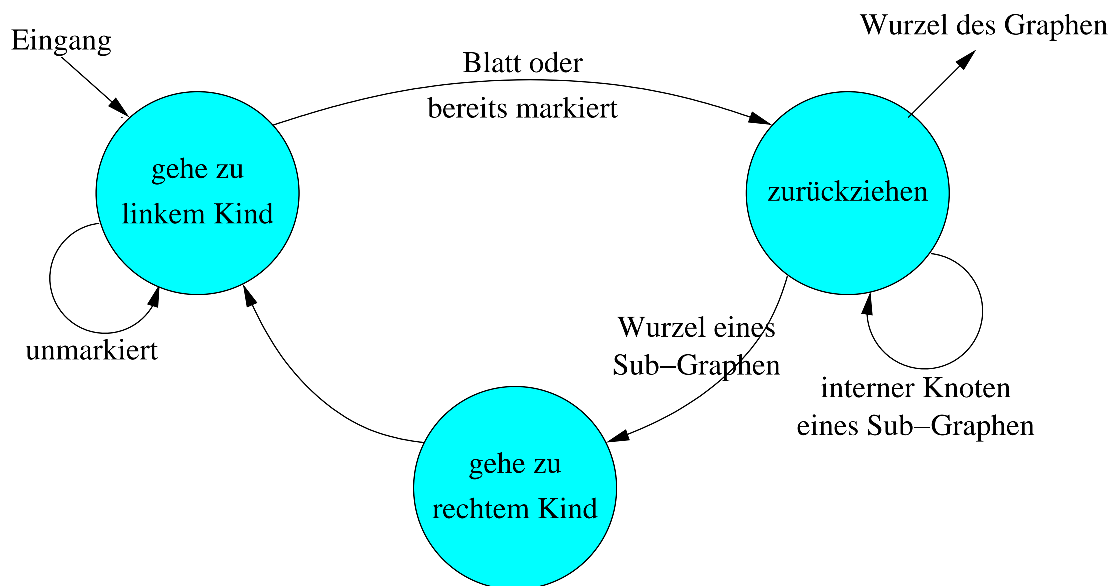

Abbildung 5: Automat zur Realisierung von Pointer Reversal

nächsten Verzweigungsknoten zurückziehen, der einen noch nicht vollständig markierten Teil-Graphen aufspannt. Dabei restaurieren wir die vormals auf die linken Kinder zeigenden Verweise. Um zu erkennen, daß der Teil-Graph vollständig, bzw. nicht vollständig markiert ist, benötigt man bei binären Graphen ein zusätzliches **Flag-Bit** für jeden Verzweigungsknoten. Diese sind zu Beginn nicht gesetzt. Erreichen wir beim zurückziehen eine Verzweigung deren Flag-Bit nicht gesetzt ist, so zeigt uns das an, daß der linke Teil-Graph vollständig markiert wurde, der rechte jedoch noch zu markieren ist.
Auf diesem Knoten wechselt der Automat dann in den Zustand *gehe zu rechtem Kind* und Phase 3 beginnt. Diese setzt das Flag-Bit und überschreibt den Verweis auf das rechte Kind mit einem Verweis auf den Vater-Knoten. Nachdem der Verweis auf das linke Kind zuvor restauriert wurde, garantiert uns dies die Erreichbarkeit der Verzweigungsknoten, die Vorfahren dieses Knotens sind. Danach wird zum ehemals rechten Kind gewechselt und der Algorithmus geht wieder in den Zustand *gehe zu linkem Kind* über, wo mit Phase 1 erneut begonnen wird.
Nachdem dieser Zyklus endliche oft durchlaufen wurde, ist schließlich auch der rechte Teil-Graph vollständig markiert, und der Algorithmus befindet sich wieder in Phase 2 im Zustand *zurückziehen*. Stößt man beim zurücklaufen auf eine Verzweigung deren Flag-Bit gesetzt ist, so zeigt dies an, daß der an diesem Knoten verwurzelte Teil-Graph vollständig markiert wurde, und der Rückzug wird fortgestzt. Im Falle eines nicht gesetzten Flag-Bits wird wie zuvor beschrieben verfahren.
Der Algorithmus terminiert, wenn er sich nicht weiter zurückziehen kann, d.h. er befindet sich wieder im Wurzel-Knoten des Halde-Graphen. In diesem Fall ist der

komplette Graph markiert.
Während des Ablaufs des Deutsch-Schorr-Waite Algorithmus' werden Zeiger umgebogen, Markierungen gesetzt und überprüft und Vorgänger bzw. Nachfolger inspiziert. Um dies zu bewerkstelligen benötigt der Algorithmus drei zusätzliche Zeiger: CURRENT, ein Zeiger der auf den aktuell inspizierten Knoten zeigt, PREVIOUS, der auf den zuvor inspizierten Knoten zeigt und NEXT, der auf den Knoten zeigt, der als nächster inspiziert wird. Damit wird unter anderem sichergestellt, daß die Struktur des Graphen beim pointer reversal erhalten bleibt. Eine genaue Beschreibung der Funktionsweise von Deutsch-Schorr-Waite findet man in [Jones and Lins, 1996]

4.2 *Pointer reversal* auf beliebigen Graphen

Für gewöhnlich sind die Strukturen auf der Programm-Halde keine binären Graphen. Ihre Knoten können, in Abhängigkeit von der Programmiersprache, beliebig viele Verzweigungen enthalten. Damit also die Deutsch-Schorr-Waite Technik überhaupt einsetzbar ist, muß sie in der Lage sein auch mit einer variablen Anzahl von Verweisen in einem Knoten umzugehen. Thorelli [Thorelli, 1972] liefert eine Methode, mittels derer man den Algorithmus entsprechend erweitern kann. Er fügt in jeden Knoten zwei zusätzliche Variablen, hier bezeichnet als n und i ein. n enthält die Anzahl der Verweise des Knotens, während i einen Zähler darstellt, der die markierten Teil-Graphen zählt, die in dem Knoten wurzeln. Oftmals benötigt man nur i, da ein entsprechendes Feld für n bereits im Kopf (Header) des Knotens kodiert ist. i hingegen muß angelegt werden und so dimensioniert sein, daß es Integer-Werte im Bereich 0 bis n speichern kann. Zu Beginn sind die i-Werte aller Knoten auf 0 gesetzt. Thorelli's Modifikation arbeitet exakt wie Deutsch-Schorr-Waite, mit dem Unterschied, daß jedesmal wenn ein Knoten K besucht wird, der i-Wert um eins hochgezählt wird. Beim ersten Besuch wird der Knoten K selbst markiert, danach und bei jedem weiteren Besuch die Markierung am i-ten Verweis fortgesetzt. Jedesmal, wenn K wieder erreicht wird, ist demnach der Teil-Graph, der vom i-ten Verweis aufgespannt wird vollständig markiert. Erreicht der Algorithmus K und stellt fest, daß $i = n$ gilt, so ist der von K aufgespannte Teil-Graph vollständig markiert und wir ziehen uns auf den Vater-Knoten zurück, wo die Markierungs-Phase entsprechend fortgesetzt wird. Sollte es keinen Vater geben, so sind wir an der Wurzel angelangt und die Markierung ist beendet.

4.3 Analyse

Auf den ersten Blick birgt Pointer Reversal große Vorteile. Wir haben die Probleme, die ein Keller mit sich bringt erfolgreich gelößt, und sieht man einmal vom Overhead der zusätzlichen Variablen in den Knoten ab, dann benötigt der Algorithmus **konstanten Platz**. Bemerkenswert dabei ist die Tatsache, daß der Keller nicht etwa aufgegeben wurde, sondern in die Knoten der Programm-Halde transferiert wurde. Einen Beweis hierzu liefert Veillon in [Veillon, 1976].
So effizient die Pointer Reversal Technik auch mit dem Platz umgeht, die Laufzeit

steigt jedoch stark an. Schorr und Waite nehmen an, daß der Algorithmus auf flachen Strukturen ca. 50 Prozent langsamer arbeitet als das iterative Markieren mit einem Keller. Diese Annahme wird von der Tatsache untermauert, daß Deutsch-Schorr-Waite einen Knoten im Verlauf des Markierens mindestens $(n + 1)$ Mal besuchen muß, wenn n die Anzahl seiner Verweise ist. Der Keller-Algorithmus hingegen kommt mit zwei Besuchen pro Knoten aus. Zusätzliche Besuche erfordern zusätzliche Speicherzugriffe. Die Wahrscheinlichkeit für einen daraus resultierenden "page fault" liegt also bei Deutsch-Schorr-Waite ungemein höher als bei der Keller-Methode und kann zu drastischen Verzögerungen führen. Und selbst bei Verfügbarkeit der Daten im Speicher oder Cache ist das Besuchen eines Knotens teurer als beim Keller basierten Markieren. Die Pointer Reversal Technik muß nämlich zusätzlich zum markieren das Flag-Bit prüfen, entsprechend setzen und vier Zeiger-Werte lesen und zurückschreiben (CURRENT, PREVIOUS, NEXT und einer der Verweise im Knoten selbst), während die Keller-Technik einfach einen Knoten vom Keller holt.
Trotz dieses schlechten Abschneidens in der Laufzeit Performance wird Pointer Reversal dennoch in einigen Systemen zur Markierung eingesetzt. Ein Beispiel dafür ist die funktionale Programmiersprache *Miranda* [Turner, 1985].

5 Markieren mit Bitmaps

Bisher bestand das Markieren von erreichbaren Knoten innerhalb eines Halde-Graphen darin ein spezielles Bit in einem erreichbaren Objekt zu setzen. Für gewöhnlich sind in den Objekten solche Bits nicht vorgesehen, so daß man zu Tricks greifen muß um eine Markierungsmöglichkeit zu haben. (Unter anderem greift man auf unbenutzte Bits zurück, die man in überdimensionierten Header-Feldern findet[1]).
Eine Alternative dazu stellt die Benutzung einer Bitmap Tabelle dar. Die Einträge einer solchen Tabelle sind einzelne Bits. Jede Adresse im Adress-Raum der Programm-Halde, die als Start-Adresse eines Objektes dienen kann, ist mit einem solchen Bit assoziiert. Ist ein Bit gesetzt, so ist das an der assoziierten Adresse beginnende Objekt markiert, andernfalls nicht. Die Größe einer Bitmap Tabelle ist äußerst gering. Implementiert als einfaches lineares Bit-Array verhält sie sich umgekehrt proportional zur Größe des kleinsten Objektes, das auf der Halde allokiert werden kann.
Mit mehreren Tabellen (z.B. für unterschiedlich große Objekte) lassen sich auf Grund der Größe der Objekte auch Bits einsparen.

Abschließend kann gesagt werden, daß Bitmaps zwei bedeutende Vorteile haben: Zum einen führt das Prüfen und Schreiben der Mark-Bits nicht mehr zu Page Faults, da wegen ihrer geringen Größe die Bitmap Tabelle komplett im Cache gehalten werden kann. Zum anderen gibt es keine Schreibzugriffe auf Halde-Objekte (gesetzt den Fall der Algorithmus kommt ohne zusätzliche Steuerbits aus) und Page Faults können nur dann auftreten, wenn Verweise dereferenziert werden. Für die Aufräum-

[1] z.B. wird der Typ eines Objektes oftmals als Integer kodiert, von dem einige Bits entbährlich sind, da die Anzahl unterschiedlicher Typen eher gering ist

Phase bedeutet das sogar, daß auf “lebendige” Objekt niemals zugegriffen wird. Diese signifikante Reduzierung von Speicherzugriffen wird jedoch teilweise von der komplexen Zugriffsart auf die Einträge in der Bitmap Tabelle wieder aufgehoben. Wenn dadurch jedoch Page Faults vermieden werden können, ist der daraus resultierende Zeitgewinn weitaus größer als der Verlust der durch die Bitmap bedingten Zugriffe.

6 Aufräumen nach Bedarf

Wenn gegen Mark-Sweep argumentiert wird, so geschieht dies meist weniger auf Grund der angesprochenen Probleme, die die Markierungs-Phase aufwirft. Einer der Hauptnachteile ist vielmehr der lineare Scan der kompletten Programm-Halde während der Aufräum-Phase. Mit den heutigen Speicher-Architekturen allerdings ist eine pauschale Analyse nicht mehr möglich. Ein linearer Scan des Speichers ist beispielsweise vorhersagbar im Vergleich zu den zufälligen Zugriffen der Markierungs-Phase oder der Kopier-Phase von *copying garbage collection.* Eine Speicher-Architektur, die *pre-fetching* unterstützt, sorgt mit dem gleichzeitigen Laden von Nachbar-Seiten der gewünschten Seite für eine Reduzierung von Page Faults in der Aufräum-Phase und wirkt sich damit positiv auf deren lineare Laufzeit aus.
Mit Bitmap Tabellen, wie sie im letzten Kapitel beschrieben wurden, wird nur noch auf die Knoten zugegriffen, die freigegeben werden sollen, da sich die zu löschenden Mark-Bits in einer Bitmap befinden, die meist komplett im Cache gehalten werden kann.
Solche speziellen Eigenheiten haben dazu geführt, daß die Laufzeit der Aufräum-Phase nur schwer oder gar nicht zu bestimmen ist. Dennoch ist sie auch weiterhin mitverantwortlich für ungewollt lange Unterbrechungen des Benutzer Prozesses. In diesem Kapitel werden daher zwei Ansätze vorgestellt, die die Unterbrechung durch die zweite Phase von Mark-Sweep verkürzen soll. Beide fallen in die sogenannte Kategorie des **lazy sweepings**.

6.1 Hughes' Lazy Sweeping

Das Prinzip von Lazy Sweep ist recht einfach. Anstatt in der Aufräum- (Sweep-) Phase sämtlichen, nicht nutzbaren Speicher freizugeben, beschränkt man sich auf eine Größe, die der Anforderung des Benutzer Prozesses entspricht. Konnte genügend Speicher zur Verfügung gestellt werden, so wird Sweep abgebrochen und erst wieder aufgenommen, wenn erneut eine Speicher-Anfrage gestellt wurde. Andernfalls muß Mark-Sweep von vorne (mit der Markierungs-Phase) beginnen.
Die Sweep-Phase wird also quasi-parallel zum laufenden Prozeß ausgeführt. Dies ist möglich, da der collector nur Objekte freigeben kann, die vom Benutzer Prozeß nicht benutzt werden können und im Gegenzug die Mark-Bits der einzelnen Knoten nicht von anderen Programmen manipuliert werden, da sie nur sichtbar für den collector sind. Lazy Sweeping reduziert also die Unterbrechungen, indem es die Kosten der Sweep-Phase auf die Speicher-Allokations-Phase transferriert.

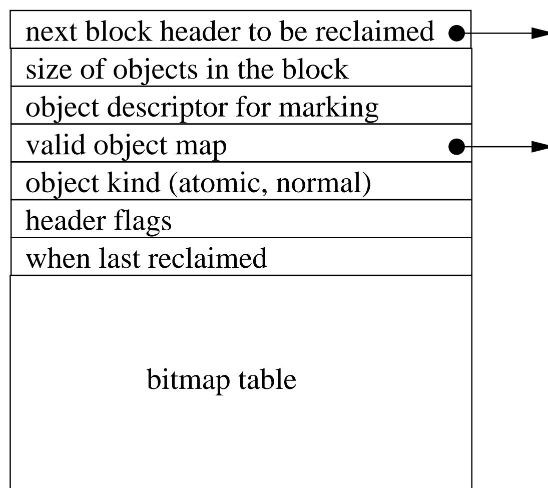

Abbildung 6: Block Header Struktur des Boehm-Demers-Weiser conservative garbage collector, Version 4.2

Ein weiterer Vorteil der direkten Übergabe von wiedergewonnenem Speicher an das anfordernde Programm ist das Wegfallen der Verwaltung dieses Speichers. Zuvor wurden recyclete Speicher-Zellen in einen speziell organisierten *pool* eingefügt und mußten von dort wieder allokiert werden.

6.2 Der Boehm-Demers-Weiser Sweeper

Dem Prinzip des "Sweepen nach Bedarf" folgt auch der Boehm-Demers-Weiser garbage collector. Auch hier wird die Sweep-Phase Stück für Stück innerhalb der Allokation ausgeführt, allerdings ist die Organisation wesentlich subtiler. Der Sweeper arbeitet auf zwei Allokations-Ebenen. Ein low-level Allokator allokiert Blöcke fester, aber konfigurierbarer Größe vom Betriebssystem (z.B. durch den Standard-Allokator MALLOC). Diese Blöcke werden dann durch einen high-level Allokator mit Objekten gefüllt. Um Fragmentierung entgegen zu wirken, sind die Objekte innerhalb der Blöcke immer gleich groß. Für jede Objekt-Größe gibt es eine sogenannte *free-list*, in der die verfügbaren Speicherfragmente nach Blöcken getrennt aufbewahrt werden.
Um die Elemente der Blöcke besser verwalten zu können, besitzt jeder Block einen separaten Block Header, wie er in Abbildung 6 schematisch dargestellt ist. Diese Header werden, nach Block Adressen geordnet, in einer verketteten Liste gehalten[2]. Ihre Informationen beinhalten unter anderem die Größe und Art (atomar oder normal) der enthaltenen Objekte und eine Bitmap Tabelle, die zur Markierung und damit zur Unterscheidung zwischen verfügbaren und nicht verfügbaren Speicherplätzen im Block dient.

[2]Es hat sich herausgestellt, daß diese Art von Organisation ein wesentlich besseres Cache-Verhalten aufweißt

Wenn nun ein Benutzer Prozeß Speicher für ein Objekt einer gewissen Größe anfordert, so schaut der high-level Allokator in der entsprechenden free-list nach. Findet er einen Speicherplatz, so wird dieser zugewiesen und aus der Liste entfernt. War die Liste leer, dann wird, sofern zuvor bereits begonnen, die Sweep-Phase wieder aufgenommen. Der nächste, von der Markierungs-Phase behandelte, aber noch nicht aufgeräumte Block wird von einer speziellen Queue (*reclaimable block queue*) geholt, die unmarkierten Elemente in die entsprechende free-list eingeordnet und die Bitmap Tabelle im Header zurückgesetzt. Dies wird so lange wiederholt, bis die entsprechende Liste nicht mehr leer ist. Daraufhin wird Sweep unterbrochen, der anfordernde Prozeß bedient und reaktiviert.
Mittels der Bitmap Tabelle in jedem Block Header kann Boehm-Demers-Weiser auch erkennen, ob ein Block vollständig geleert wurde. In diesem Fall wird der Block an den low-level Allokator zurückgegeben. Wenn die Sweep-Phase terminiert, aber nicht genügend Speicher zur Verfügung stellen konnte wird entweder, falls dies nicht schon zuvor geschen ist, ein neuer garbage collection Prozeß begonnen, andernfalls wird die Halde vom low-level Allokator durch einen neuen Block erweitert.

7 Vorteile und Nachteile

Der Mark-Sweep garbage collector offeriert ein paar bemerkenswerte Vorteile. Seine Fähigkeiten, zyklische Datenstrukturen ohne zusätzlichen Aufwand zu behandeln, den verfügbaren Speicher nahezu vollständig den Benutzer Prozessen zur Verfügung zu stellen[3] und das simple Interface zwischen Benutzer Programm und collector (keine Verwaltung von Verweisen), verschafft ihm einige Attraktivität gegenüber anderen Kollektoren. Wenn Bitmap Tabellen benutzt werden, kann auch eine Menge Platz gespart werden, der zuvor architekturbedingt bis zu einer Größenordnung von Bytes pro Knoten zur Markierung benötigt worden ist. Die Lokalität der Mark-Bits in einer Tabelle sorgt außerdem für ein besseres Cache Verhalten, da sie vollständig im Cache gehalten werden kann und Zugriff auf mehrere Mark-Bits mit einer einzelnen Instruktion möglich sind.
Dennoch besitzt Mark-Sweep auch Nachteile: Ohne zusätzliche Erweiterungen des Algorithmus' muß mit zunehmender Fragmentierung des Speichers gerechnet werden, da unerreichbare Speicherzellen nur freigegeben, nicht aber, wie bei Copying Collection, neu strukturiert werden. In *real memory* Systemen kann dies zu Performance Einbußen führen, in *virtual memory* Systemen gar zu exzessivem "thrashing" und damit zum beinahen oder völligen Stillstand. Abgesehen davon hängen die garbage collection Intervalle von der Größe des wiedergewonnenen Speichers ab. Bei starker Fragmentierung kann jedoch der davon brauchbare Teil gleich Null sein. Daraus resultiert, daß die Frequenz der collections mit der Verweildauer des Benutzer Programms im Speicher zunimmt.
Ein weiterer Punkt ist die Unterbrechung des Benutzer Prozesses für die Dauer einer kompletten garbage collection. Zwar kann die Zeitspanne der Unterbrechung durch

[3] Im Gegensatz zu *Copying Collection*

lazy sweeping (s. Kapitel 5) verkürzt werden, dies ändert allerdings nichts an der Tatsache, daß Mark-Sweep in den meisten Fällen nicht praktikabel für "real-time", stark interaktive oder verteilte Systeme ist. Wenn es darauf jedoch nicht ankommt, ist seine Performance weitaus besser als die eines inkrementellen garbage collectors. Ein Vergleich mit einem copying collector ist da schon schwieriger. Da die Sweep-Phase auf die Allokation aufgeteilt werden kann, stellt sich im wesentlichen die Frage, was ist teurer, markieren (Mark-Sweep) oder kopieren (Copying)? Wenn man einmal von kleinen Objekten absieht, ist es einfacher Objekte zu markieren als sie zu kopieren. Auf der anderen Seite ist die Allokation von Speicher mit Mark-Sweep teurer als mit Copying Collection.
Abschließend kann nur gesagt werden, daß es bei den vielen Varianten der einzelnen garbage collection Techniken nicht mehr möglich ist ein pauschales Urteil über ihre Eignung abzugeben. Am Ende hängt die Wahl für den "besten" Algorithmus stark davon ab, wie sich ein Benutzer Programm auf der Halde verhält und ob die collection time die allocation time dominiert. Es hat sich dabei herausgestellt, daß Mark-Sweep des öfteren sehr gute Ergebnisse erziehlt.

Literatur

[Boehm and Weiser, 1988] Hans-Juergen Boehm and Mark Weiser. Garbage collection in an uncooperative environment. *Software Practice and Experience*, 18(9):807-820, 1988.

[Knuth 1973] Donald E. Knuth. *The Art of Computer Programming*, volume I: *Fundamental Algorithms*, chapter 2. Addison-Wesley, second edition, 1973.

[Jones and Lins, 1996] Richard Jones and Rafael Lins. *Garbage collection: algorithms for automatic dynamic memory management*, John Wiley & Sons Ltd, 1996, pages 75-95

[Schorr and Waite, 1967] H. Schorr and W. Waite. An efficient, machine independent procedure for garbage collection in various list structures. *Communications of the ACM*, 10(8):501-506 August 1967

[Thorelli, 1972] Lars-Erik Thorelli. Marking algorithms. *BIT*, 12(4):555-568, 1972

[Turner, 1985] David A. Turner. Miranda - a non-strict functional language with polymorphic types. In Jouannaud [FPCA 1985], pages 1-16

[Veillon, 1976] G. Veillon. Transformations de programmes recursifs. R.A.I.R.O. *Informatique*, 10(9):7-20, September 1976.