Mark and Sweep

Seminar Softwareentwicklung: Garbage Collection

Eva Schartner, Studentin, JKU Linz, eva.schartner@students.jku.at

Abstract

Garbage Collection ist eine der angenehmeren Neuerungen beim Programmieren, vor allem aber ist sie eine der nützlichsten für den Programmierer. Mit ihrer Einführung musste man sich nicht mehr selbst um die Bereinigung des Speichers kümmern, was eine große Arbeitserleichterung darstellte. Für moderne Programmiersprachen ist Garbage Collection schon so selbstverständlich, dass bei Lehrveranstaltungen der Begriff nur noch erwähnt wird, jedoch weder auf die Arbeitsweise noch auf die Hintergründe von Garbage Collection eingegangen wird. Dadurch sind auch die verschiedenen Wege, wie Speicherbereinigung funktionieren kann, nicht mehr im generellen Wissensrepertoire von Informatikern vorhanden. Eine der Möglichkeiten, wie Garbage Collection arbeitet, werde ich im folgenden genauer vorstellen: Den Mark and Sweep Algorithmus. Erst werde ich seinen prinzipiellen Ablauf erklären, danach Stärken und Schwächen erläutern. Dies auch im Hinblick auf andere Garbage Collection Verfahren. Nach einer genaueren Betrachtung von Mark and Sweep spezifischen Merkmalen werde ich noch auf jene Weiterentwicklungen des ursprünglichen Mark and Sweep Algorithmus eingehen, die sich eine Steigerung der Performanz zum Ziel gesetzt haben.

I. Einführung

Kein Programmierer wird sich heutzutage beklagen, wenn er mit einer Programmiersprache arbeitet, die Garbage Collection zur Verfügung stellt. Man braucht sich nur einen kurzen Augenblick das Schreiben von Programmen in C/C++ vor Augen zu führen. Hier gibt es kein automatisches Verfahren, das den Speicher bereinigt, und der Programmierer muss sich daher selbst darum kümmern. Nicht allein die zusätzliche Schreibarbeit ist lästig, die Hauptanstrengung ist das ständige Mitdenken, das Entscheiden, ob eine Datenstruktur in Zukunft noch gebraucht wird oder man ihren Platz im Speicher für andere Daten nutzen kann. Ein Garbage Collector entscheidet nicht nur über dieses Problem, er läuft auch ohne äußere Einflüsse, etwa den expliziten Befehl eines Programmierers, an. Nun gibt es verschiedene Wege, wie ein solcher Garbage Collector realisiert werden kann. Einige bekannte Namen, die in der Anfangsphase der Garbage Collector Entwicklung häufig in der Literatur auftauchen, sind Reference Counting, Mark and Sweep und Copying Collectors.

Reference Counting zählt, wie aus dem Namen leicht herauszulesen ist, die Referenzen, Pointer, die auf jede Zelle im Speicher verweisen. Sinkt diese Ziffer auf Null, so wird die Zelle als löschbar, sogenannter garbage, angesehen. Bei der Copying Collector Methode wird der Heap in zwei Teile geteilt, sogenannte Semi-Spaces: den From-Space und den To-Space. Aktive Objekte werden beim Garbage Collection Vorgang zwischen den beiden Teilen kopiert.

Mark and Sweep ist ein Algorithmus, der zum ersten Mal in den 60er Jahren in der Literatur auftaucht. Hervorgegangen ist er aus einer Überlegung McCarthy's heraus, der Garbage Collection bei der Lisp-Implementierung verwendete. Seit damals haben sich Garbage Collection Algorithmen natürlich weiterentwickelt.

Mark and Sweep ist ein sogenannter Tracing Algorithmus. To trace, aus dem Englischen übersetzt, bedeutet aufspüren, nachspüren, verfolgen. Damit ist die Grundidee von Mark and Sweep auch schon erklärt. Ausgehend von der Menge der Wurzeln des laufenden Programms werden alle Zeiger, die von ihnen wegführen, verfolgt. Alle Zellen im Speicher, die bei diesem Vorgang angetroffen werden, gelten als aktiv, lebendig. Im Gegensatz dazu stehen jene Zellen, die bei dieser Traversierung nicht besucht wurden. Auf sie existiert keine Referenz mehr, ihr Inhalt kann nicht mehr vom Programm abgefragt werden. Sie gelten als unerreichbar, inaktiv. Beim nachfolgenden Prüfen aller Heapeinträge werden alle jene Zellen, die nicht aktiv sind, als garbage eingesammelt.

Ein Merkmal eines Garbage Collectors nach Mark and Sweep Prinzip ist die Tatsache, dass er erst dann zu laufen beginnt, wenn kein freier Speicher mehr vorhanden ist. Solange die eigentliche Anwendung noch freien Speicher findet, wird der Garbage Collector nicht aufgerufen. Er verbraucht somit keine Ressourcen des Systems, sofern es nicht absolut notwendig ist.

Eine Konvention in der Literatur, die auf Dijkstra zurückgeht, ist das Zuweisen von Farben. Aktive Zellen werden als schwarz angesehen, zu bereinigende Zellen als weiß.

Eine weitere Klärung: Je nach Betrachtung seiner Rolle wird das reguläre Programm, also alles was nicht Garbage Collector ist, in der Literatur auch als Evaluator oder Mutator

bezeichnet. Ich werde die Begriffe im folgenden wechselweise verwenden.

Im Anschluss werde ich nun den oben kurz geschilderten Vorgang im Detail erläutern. Alle Algorithmen, die in der Folge präsentiert werden, sind im Pseudocode gehalten. Da sie über die Jahre hinweg in Hinblick auf verschiedene Programmiersprachen entwickelt wurden, die nicht mehr allgemein geläufig sind, wird so das Verständnis erleichtert.

II. Funktionsweise von Mark and Sweep

Beim Anlegen einer neuen Datenstruktur, wie auch immer diese beschaffen sein mag, wird ein Allozierungsvorgang gestartet. Hierbei wird der Speicher nach einem freien Block durchsucht, der groß genug ist, das neue Datum aufzunehmen. Freie Blöcke werden, abstrahiert betrachtet, in einem sogenannten „free_pool“ gehalten. Für den Fall, dass kein solcher Block auffindbar ist, wird vom System der Garbage Collector gestartet. Dieser bereinigt den Speicher in der Regel so, dass eine neue Zelle alloziert werden kann, die an den Aufrufer zurückgegeben wird. Im untenstehenden Algorithmus ist dies zu sehen.

```
New() =
        if free_pool is empty
                mark_sweep()
        newcell = allocate()
        return newcell
```

Algorithmus 1: Allocate, [3]

Garbage Collection, die nach Mark and Sweep Algorithmen abläuft, zeichnet sich durch 2 Phasen aus: Einer Markierungsphase, Mark, und einer Bereinigungsphase, Sweep. Da Sweep auf die Ergebnisse der Markierungsphase angewiesen ist, müssen diese beiden Phasen sequentiell ablaufen. Sweep kann seine Funktion, nicht mehr erreichbare Zellen dem free_pool hinzuzufügen, ohne die von Mark hinterlassenen Markierungen, nicht nachkommen. Aus dem folgenden, sehr einfach gehaltenen, Algorithmus ist die Reihenfolge ersichtlich.

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

Algorithmus 2: Mark and Sweep, [3]

Für jede Wurzel wird mark(Wurzel) aufgerufen. Erst wenn jeder dieser Aufrufe zurückgekehrt ist, kommt sweep() zum Zug. Ist nach dem Ende von sweep() genügend freier Speicher vorhanden, so wird das normale Programm fortgesetzt, da allocate nun die benötigten Ressourcen zur Verfügung stehen, um neue Objekte anzulegen. Aber natürlich darf man das Worst-Case Szenario nicht außer Acht lassen. Im schlimmsten Fall konnte die Garbage Collection Routine den free_pool nicht mit neuen Speicherzellen versorgen und das Anlegen einer neuen Datenstruktur nicht zu Ende gebracht werden. Hier bleibt einem die Wahl zwischen einem Programmabbruch mit der Fehlermeldung, dass der Speicher voll ist, und einem Erweitern des Heaps, um der Anwendung mehr Speicher zur Verfügung zu stellen. [3]

Wie sehen nun die Markierungs- und die Bereinigungsphase im Detail aus?

A. Mark

Mark hat die Aufgabe, aktive Zellen zu identifizieren, also jene, die von einer Wurzel aus erreicht werden können. Solange noch eine Referenz auf die Zelle vorhanden ist, hat das Programm noch eine Verwendung für ihren Inhalt. Diese Referenzen, von der Wurzel zu einem anderen Speicherobjekt zu einem weiteren Objekt in Speicher usw. bilden sozusagen den Pfad, dem Mark folgt. Bei jedem Objekt, auf das der Algorithmus trifft, hinterlässt er eine Markierung. In der Literatur ist hier der Ausdruck „markbit“ sehr geläufig. Es handelt sich also um ein Bit des Objekts, das eine spezielle Bedeutung erhält. Wird es auf 1 gesetzt, so ist die Speicherzelle aktiv, 0 bedeutet nicht aktiv. Wieviel Speicherplatz dieses Markbit aber benötigt ist implementationsabhängig. Wird in der bestehenden Struktur ein Bit gefunden, das als Markbit verwendet werden kann, so hat man keine zusätzlichen Speicherkosten. Dies ist natürlich nicht immer möglich, in welchem Fall für die Markierung zusätzlicher Speicher anfällt: im besten Fall ein Bit, im schlimmsten Fall ein Byte oder ein Word. [3]

Bei Zellen, auf die keine Referenz mehr existiert, wird diese Markierung nicht gesetzt, da Mark sie ja gar nicht erreicht. Dieser Unterschied ermöglicht es der nachfolgenden Sweep-Phase zu erkennen, ob ein Knoten noch vom Programm gebraucht wird.

In Pseudo-Code sieht das folgendermaßen aus:

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

Algorithmus 3: Mark, [3]

Der oben beschriebene Pfad, der von den Referenzen gebildet wird, ist natürlich nur im Idealfall linear. Mit an Sicherheit grenzender Wahrscheinlichkeit wird man auf Zellen stoßen, die mehr als nur einen Nachfolger haben. Beim Auftreten einer Verzweigung folgt Mark einem dieser Subpfade bis zu seinem Ende. Anschließend kehrt Mark wieder zu N, dem Verzweigungspunkt zurück, um den nächsten Subgraphen zu traversieren. Beim Antreffen von weiteren Verzweigungspunkten wiederholt sich dieser Vorgang. Jedesmal, wenn Mark einer Verzweigung folgt, muss er sich natürlich die Adresse der Verzweigungszelle merken, um am Ende des Subgraphen wieder hierher zurückkehren zu können. Wie man dies geschickt macht, darauf werde ich in Kapitel 3 eingehen.

Wie aus obigem Algorithmus ersichtlich, kann Mark auf eine Zelle stoßen, deren Markbit schon gesetzt worden ist. Dies ist möglich, wenn auf diese Zelle mehr als eine Referenz verweist; Mark also schon beim Verfolgen eines anderen Pointers auf die Zelle gestoßen ist. In Algorithmus 3 wird in so einem Fall das Durchwandern des Subgraphen abgebrochen. Denn das gesetzte Markbit bedeutet ja gerade, dass Mark schon hier gewesen ist und in der Folge auch jenen Referenzen gefolgt ist, die von N wegführen. Es gibt noch einen weiteren Grund, warum diese Eigenschaft absolut notwendig ist: im Fall von zyklischen Datenstrukturen. Betritt Mark eine solche, so markiert er alle Objekte bis er wieder am Eintrittsobjekt ankommt. Ohne die Abbruchbedingung des schon gesetzten Markbits würde Mark nie mehr den Zyklus verlassen.

B. *Sweep*

Dass die Bereinigung erst nach dem Ende von Mark beginnen kann, habe ich oben schon erklärt. Ein weiterer Unterschied zwischen den beiden Phasen ist die Tatsache, dass Mark seine Arbeit an den Wurzeln beginnt und lediglich aktive Zellen besucht. Sweep hingegen durchläuft den Heap linear. Wie an folgendem Algorithmus zu sehen, beginnt der Durchlauf am unteren Ende und endet an der oberen Grenze des Heaps. Dies bedeutet, dass jede Zelle von Sweep besucht und ihr Markbit begutachtet wird.

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

Algorithmus 4: Sweep, [3]

Trifft Sweep auf eine weiße Zelle am Heap, so erkennt es diese als unerreichbar; die Zelle kann somit zum Pool freier Zellen hinzugefügt werden. Wie free(N) genau funktioniert ist abhängig von der Implementierung des free_pools. Meist ist in der Literatur von einer Liste die Rede, der „free_list“, die alle freien Speicherzellen miteinander verknüpft. Ein Vorteil dieser Variante sind die geringen Kosten, die in diesem Fall für free(N) anfallen würden. Um N an die free_list hängen, genügt es, einen Zeiger von der Liste auf N zu setzen. Andere Implementierungen sind jedoch genauso zulässig. [3]

Ist hingegen das Markbit der aktuell von Sweep geprüften Zelle gesetzt, so darf diese Zelle nicht gelöscht werden, da ihr Inhalt vom Programm noch gebraucht wird. Sweep setzt daher lediglich das Markbit zurück. Der Grund hierfür ergibt sich aus folgendem Beispiel:

Angenommen, das Markbit einer aktiven Zelle Z würde nicht zurückgesetzt. Nach dem Ende des Garbage Collectors ist das reguläre Programm wieder aktiv. Im Laufe seiner Berechnungen erkennt es, dass es Z nicht mehr benötigt und löscht die Referenz auf Z, deren Markbit immer noch schwarz ist. Ein Zurücksetzen des Markbits ist aber auch nicht mehr möglich, da ja auf Z keine Referenz mehr existiert. In der Folge kann Z nie von Sweep freigegeben werden, und blockiert bis zum Ende der Programmlaufzeit Speicherplatz. Das Löschen des Markbits ermöglicht es der nächsten Bereinigungsphase, die Zelle zu löschen, falls sie bis dahin zu garbage geworden ist.

Durch das lineare Durchlaufen des Heaps kennt Mark and Sweep jene Probleme mit zyklischen Datenstrukturen nicht, die für Reference Counting ein großes Hindernis sind, konkret den Fall, dass sich die Objekte des Zyklus' gegenseitig referenzieren, obwohl die externe Referenz schon verlorengegangen ist.

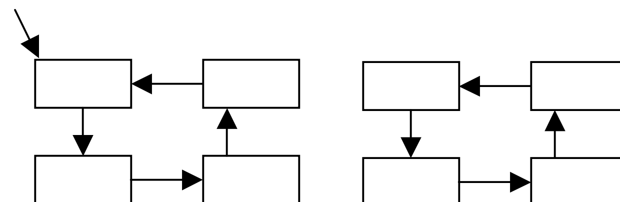


Abbildung 1: zyklische Struktur mit und ohne externe Referenz

Da Sweep jede Zelle besucht, und nur deren Erreichbarkeit von einer Wurzel aus beurteilt wird, ist keine Sonderbehandlung von Zyklen nötig. Referenzen innerhalb eines Zyklus' fallen bei diesem Verfahren nicht negativ auf. [3]

III. Stärken und Schwächen von Mark and Sweep

Nachdem die grundlegende Funktion von Mark and Sweep nun erklärt wurde, möchte ich im folgenden die Stärken nur kurz erwähnen, da diese in Kapitel II schon genauer beschrieben wurden. Ich werde aber detaillierter auf Schwächen eingehen, auch im Hinblick auf jene von Reference Counting.

A. *Stärken*

Wie oben schon erwähnt, werden Zyklen von Mark and Sweep ohne zusätzlichen Aufwand behandelt. Dies ist ein Grund, warum Systeme, deren Garbage Collector auf Reference Counting basiert, als Back up Methode Mark and Sweep einsetzen, zum Beispiel Modula-2+.

Weiters fällt kein Overhead für das Verschieben von Zeigern an, wenn eine Speicherzelle in die free_list eingehängt wird.

Ein Mark and Sweep Garbage Collector wird nur aktiv, wenn kein freier Speicherplatz mehr für Allokationen gefunden wird. Dies erspart CPU-Belastung gegenüber Reference Counting, wo ständig die Anzahl der eingehenden Referenzen einer Speicherzelle berechnet wird. [3]

B. *Schwächen*

Der große Nachteil von Mark and Sweep liegt in der Tatsache begründet, dass die beiden Phasen den Heap

durchlaufen müssen. Während dieses Vorgangs darf das Hauptprogramm selbstverständlich nicht auf den Heap zugreifen und dort gespeicherte Zelleninhalte ändern beziehungsweise neue Zeiger setzen, da sonst Mark und Sweep falsche Ergebnisse liefern können. Man stelle sich vor, Mark hat eine Speicherzelle A als aktiv markiert. Während Mark noch läuft, löscht das reguläre Programm die Referenz auf A. Der nachfolgende Sweep sieht nur das gesetzte Markierungsbit von A und fügt es daher nicht dem free_pool hinzu. Da die Referenz auf A aber nicht mehr existiert, kann A nie mehr weiß werden und bleibt somit als Speicherleiche zurück.

Auf die Möglichkeit, mit geeigneter Synchronisation Mark, Sweep und den Mutator gleichzeitig ablaufen zu lassen, darauf gehe ich in Kapitel 6 und 7 ein.

Bei Mark and Sweep handelt es sich um einen sogenannten Start/Stop-Algorithmus. Bevor Mark anlaufen kann, muss das reguläre Programm angehalten werden. Erst nach Beenden von Sweep kann es wieder fortsetzen. Abhängig von der Dauer des Garbage Collectors kann das eine nicht unbeträchtliche Beeinträchtigung der Performanz zur Folge haben. So verbrachten Lisp-Programme in den frühen 1980er Jahren 25% - 40% ihrer Zeit damit, den Speicher zu bereinigen; für die Benutzer hatte das eine Wartezeit von 4.5 Sekunden alle 79 Sekunden zur Folge.

Daher lässt sich ableiten, dass Mark and Sweep Verfahren nicht für Echtzeitsysteme geeignet sind. Man stelle sich eine sicherheitskritische Anwendung vor, die einen signifikanten Teil der Ausführungszeit für den Garbage Collector verwendet, dessen Ausführungszeitpunkt ja nicht vom Programmierer festgelegt wird und somit zu jedem beliebigen Zeitpunkt stattfinden kann.

Mark and Sweep verursacht hohe Zeitkosten. Während der Markierungsphase wird jede aktive Zelle besucht, im schlimmsten Fall ist das der gesamte Heap. Noch schlimmer steht es mit der Sweep-Phase, in der immer jede Speicherzelle überprüft wird. Daher ist die Komplexität von Mark and Sweep direkt proportional zu der Größe des Heaps. Im Gegensatz hierzu stehen Copying Collectors, die lediglich direkt proportional zur Größe der aktiven Daten sind. [8]

Auch die zusätzlichen Speicherkosten, die anfallen wenn kein geeignetes unbenutztes Bit für das Markbit gefunden wird, können beträchtlich sein. Dadurch kann es passieren, dass Mark and Sweep ebensoviel zusätzlichen Speicher pro Knoten braucht, wie es beim Reference Counting der Fall ist.

Ein damit verwandtes Problem ist durch die Wirkungsweise des Mark and Sweep Garbage Collectors begründet. Je mehr Speicherzellen des Heaps belegt sind, desto länger dauert eine Garbage Collection Phase, da diese ja abhängig ist von der Größe des Heaps. Gleichzeitig wird der Pool jener Zellen, der für neue Allokationen zur Verfügung steht, immer kleiner. Daher ist es wahrscheinlich, dass der Garbage Collection Vorgang öfter aufgerufen wird, der weniger freie Zellen zurückbringt und gleichzeitig immer länger dauert. Der Mutator kommt immer seltener an die Reihe. Um ein solches „thrashing“ des Garbage Collectors zu verhindern, ist eine genügend große Anzahl von freien Speicherzellen im Heap nötig. Reference Counting kennt diese Limitation nicht.

Hängt man erst einmal in eben beschriebenem Teufelskreis, kann man nur ausbrechen, indem man den Heap selbst vergrößert. Dies kann jedoch noch mehr Probleme verursachen, da als Konsequenz die Zellen des Heaps auf verschiedene Pages und in weiterer Folge über den Speicher und die Platte verteilt würden. Mehr page faults wären die Folge, sodass die ohnehin schon schlechte Performanz des Mutators (der ja fast nicht mehr zum Zug kommt) nur noch weiter abnehmen würde.

Mark and Sweep hat die Angewohnheit, den Heap zu fragmentieren. Um dem entgegenzuwirken, wird oft noch eine sogenannte Kompaktierungsphase nachgeschaltet. Diese soll aktive Zellen im Heap verdichten, sodass das Auffinden geeigneter freier Blöcke zur Allozierung vereinfacht wird. Hierbei fällt jedoch wieder Overhead an, da Objekte von einem Speicherplatz zu einem anderen verschoben werden müssen. [8], [3]

IV. Implizite Rekursion und Marking Stack

Trifft Mark während seines Durchlaufs auf eine Verzweigung, so muss er sich diesen Verzweigungspunkt merken, um vom Ende eines Astes wieder hierher zurückkehren zu können. Der vorgestellte Algorithmus ist rekursiv. Geht man von einem reellen Anwendungsbeispiel aus, wird dieses viele Verzweigungen, mit einer teils großen Anzahl von Ästen, enthalten. Läuft Mark nun über so eine Datenstruktur, so wird sich eine nicht zu unterschätzende Anzahl von Verzweigungspunkten ansammeln; während Mark also einem Pfad folgt, wird die Adresse jeder Verzweigung im Systemstack gespeichert, um ein späteres Zurückspringen zu ermöglichen. Bei komplexen Strukturen kann dies dazu führen, dass ein Overflow auftritt und die gesamte Berechnungstätigkeit zum Stillstand kommt.

Als Lösung dieses Problems bieten sich zwei Methoden an: Die Rekursion in iterative Schleifen aufzulösen und einen Hilfsstack, den sogenannten MarkStack, zur Speicherung der Verzweigungspunkte, zu verwenden. [3]

A. Algorithmus

Durch die explizite Speicherung der Verzweigungspunkte erhält man mehr Kontrolle über die Markierungsphase. Die Tiefe des Stacks ist gleich der Länge des längsten Referenzpfads; im schlimmsten Fall sieht man auf den ersten Blick keinen Vorteil gegenüber der rekursiven Methode. Ein Überlauf kann zwar immer noch auftreten, allerdings kann man jetzt aktive Maßnahmen ergreifen, um dieses Problem zu lösen.

Im nachfolgenden Algorithmus gilt für Sweep noch der oben genannte Algorithmus, da die eben besprochenen Maßnahmen keine Auswirkungen auf Sweep haben.

```
gc() =
    mark_heap()
    sweep()
```

```
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)
```

Algorithmus 5: Iteratives Mark and Sweep mit MarkStack, [3]

mark_heap() beschreibt die Initialisierungsphase des Algorithmus. Am Anfang ist der MarkStack leer. Jede Wurzel wird markiert und anschließend auf den Stack gelegt.

mark() holt sich die Wurzel vom Stack und terminiert erst, wenn kein Knoten (= Verzweigungspunkt) mehr auf dem Stack liegt und die Datenstruktur somit vollständig markiert worden ist. Jedes Kind des aktuell betrachteten Knotens wird markiert. Handelt es sich bei der im Moment behandelten Zelle M um keine atomare, das heißt enthält sie mindestens eine Referenz auf eine andere Speicherzelle, so wird auch M auf den MarkStack gelegt, damit von ihr ausgehenden Referenzen bei einem anderen while-Durchlauf nachgegangen werden kann.

So kommen in untenstehender Datenstruktur die Knoten 1, 2, 3 und 5 auf den Stack (natürlich 1 nicht gleichzeitig mit den anderen), 4 und 6 hingegen nicht.

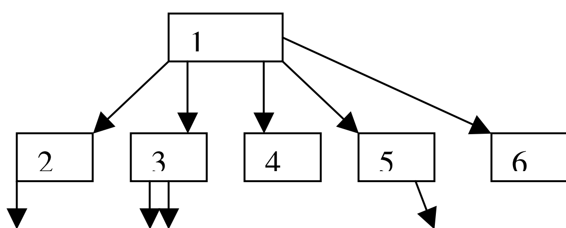


Abbildung 2: Datenstruktur mit atomaren Knoten

Eine Schwäche des obigen Algorithmus ist die Tatsache, dass der zuletzt gepushte Knoten, sowohl von mark_heap() als auch von mark(), sofort mit der ersten Anweisung der while-Schleife wieder vom Stack genommen wird. Als Optimierung könnte man sofort der Referenz der letzen Zelle folgen und sich so zwei Operationen sparen sowie die Tiefe des Stacks um 1 reduzieren. [3]

Diese letzte Überlegung ist nicht zu vernachlässigen, da ein MarkStack-Overflow nicht ausgeschlossen werden kann.

B. *Overflow*

Der Algorithmus muss im Falle eines Stack-Überlaufs in der Lage sein, in einen gültigen Systemzustand zurückzukehren, sowie den Markierungsvorgang erfolgreich abzuschließen. Um einen Stack Overflow zu vermeiden, schlagen Boehm und Weiser vor, große Arrays nicht in einem Stück, sondern in Gruppen auf den MarkStack zu legen. [1]

Was kann man aber nun unternehmen, wenn der MarkStack übergelaufen ist?

Als erstes ist es wichtig, den Überlauf zu erkennen. Einerseits kann man den Stack selbst beobachten oder die letzte page des Stacks mit einer Schreibsperre versehen, sodass bei vollem Stack eine Exception geworfen wird, die man abfangen und entsprechend behandeln kann. [3]

Hat man den Überlauf erkannt, stellt sich die Frage: Was tun?

Boehm, Demers und Weiser pushen in so einer Situation keine Verzweigungspunkte mehr: Zuerst wird zur Kenntnis genommen, dass ein Überlauf aufgetreten ist. Die Markierungsphase wird nicht angehalten sondern läuft ganz normal nach Algorithmus 5 weiter. Allerdings werden keine Push-Operationen mehr durchgeführt. Da nur noch Pop-Operationen möglich sind, arbeitet Mark langsam aber sicher den vollen Stack ab. Wenn der MarkStack wieder leer ist, geschehen zwei Dinge. Erstens wird ein neuer Stack von doppelter Größe angelegt. Dies geschieht als Vorsichtsmaßnahme, da der alte Stack erwiesenermaßen zu klein war. Außerdem beginnt der Garbage Collector jetzt den Heap nach Einträgen zu durchsuchen, die zwar selbst markiert sind, deren Kinder aber noch weiß sind. Zur Erinnerung: Trifft Mark auf einen aktiven Knoten, markiert er diesen sofort und legt ihn danach auf den Stack, damit in späterer Folge auch dessen Kinder markiert werden können. Die nicht durchgeführten Push-Operationen müssen nun also nachgeholt werden. [1], [3]

Knuth hingegen sieht den Stack als zyklische Datenstruktur an. Er legt Pointer auf den MarkStack modulo h, wobei h die Größe des Heaps ist. Zur Veranschaulichung: Angenommen, h =5, der Heap kann also fünf Einträge aufnehmen. Die sechste push-Operation würde eigentlich einen Overflow auslösen, 6 modulo 5 ergibt jedoch 1.

Knuth überschreibt somit den ersten Eintrag des Stacks mit dem aktuellen (sechsten) Verzweigungspunkt. Es kann somit nicht mehr garantiert werden, dass jene Kinder des ursprünglichen ersten Knotens von Mark besucht worden sind. Knuth löst dieses Problem, indem er den Heap scannt, sobald der MarkStack leer geworden ist. Er such ebenso wie Boehm, Demer und Weis nach Knoten, die selbst zwar markiert wurden, deren Kinder aber noch unmarkiert sind. Er legt solche Knoten wieder modulo h auf den Stack, wartet bis der Stack leer ist usw. Am Ende tritt der Fall auf, dass der MarkStack leer ist, und keine aktiven Knoten Zeiger auf nicht aktive enthalten; der Markierungsvorgang ist somit abgeschlossen. [4]

V. Speicherung der Markierung

Ein Problem, das ich schon angesprochen habe, ist es, einen Speicherplatz für die Markierung, das Markbit, zu finden. Bis jetzt wurden nur Fälle erwähnt, wo dieses Zeichen direkt in

der Speicherzelle gehalten wurde. Aber natürlich geht es auch anders.

Eine effiziente Weise, das Markbit zu speichern, erreicht man durch die Verwendung einer separaten Bitmap. Jede Speicherzelle, die den Beginn eines Objekts enthalten kann, wird über seine Adresse genau einem Bit der Map zugeordnet. Da hier garantiert nur ein Bit für die Markierung verwendet wird, ist dies eine sehr speicherschonende Lösung. Ist die Bitmap als lineares Feld von aufeinanderfolgenden Bits implementiert, so ist die Menge an Speicher, die sie in Anspruch nimmt, indirekt proportional zur Größe des kleinsten Objekts, das im Heap alloziert werden kann. Das Bit, das zu einer Adresse a gehört, lässt sich einfach durch eine Shift-Operation ermitteln.

Eine weiterentwickelte Variante würde für jede Art von Objekten eine eigene Bitmap verwenden, der Zugriff erfolgt hierbei unter Verwendung einer Hashtable oder eines Suchbaumes. Der Vorteil hierbei ist, dass nicht jede Adresse im Heap mit einem Markbit assoziiert werden muss, da große Objekte mehr als eine Speicherzelle belegen.

Ein Vorteil im Abspeichern des Markbits in einer Bitmap ist durch deren geringe Größe bedingt. Dadurch kann sie im Arbeitsspeicher gehalten werden und das Lesen oder Schreiben von Markbits würde nie zu einem page fault führen. Das wirkt sich natürlich positiv auf die Performanz des Garbage Collectors aus.

Ein weiterer Vorteil von Bitmaps wird während der Ausführung von Sweep ersichtlich. Es müssen nun lediglich die Bits der Bitmap geprüft und, wo nötig, zurückgesetzt werden. Die Objekte am Heap werden von Sweep nicht mehr angesehen. Einzig das Verlinken der inaktiven Knoten in die free_list greift noch auf den Heap zu.

Ein Nachteil von Bitmaps sind die erhöhten Kosten die anfallen, wenn die Adresse im Heap auf das Markbit gemappt wird, speziell wenn sowohl der Heap als auch die Bitmap nicht zusammenhängend sind. Dieser Aufwand fällt natürlich nicht an, wenn das Markbit direkt im Objekt gespeichert wird.

[3]

VI. Zeigerumkehr

Eine andere Möglichkeit, um den Speicherverbrauch des Garbage Collectors zu senken, befasst sich nicht mit den Markbits, sondern mit den Verzweigungspunkten. Wie in Kapitel III schon erläutert, muss für jede Verzweigung die Rückkehradresse auf einem Stack hinterlegt werden, wodurch der ohnehin schon knapp bemessene freie Speicher weiter verkleinert wird. Das Verfahren der Zeigerumkehr geht in eine ganz andere Richtung.

Die Grundidee ist folgende: Anstatt die Adresse des Verzweigungspunkts zu speichern, dreht man alle Zeiger, die man auf dem Weg hinab entlanggewandert ist, um. Nun zeigt nicht mehr der Elternknoten auf den Kindknoten, sondern das Kind enthält die Referenz auf seinen Vater. Dies ermöglicht es dem Garbage Collector, einfach denselben Weg wieder zurückzugehen, diesmal aufwärts. Natürlich werden bei dieser Rückwärtstraversierung die ursprünglichen Zeiger wiederhergestellt, da die Datenstruktur selbst ja nicht verändert werden soll.

Ein Algorithmus, der das ermöglicht, ist der Deutsch-Schorr-Waite Algorithmus.

```
mark(R)=
   done = false
   current = R
   previous = null
   while not done
   // follow left pointers
      while current <> null
      and mark_bit(current) == unmarked
         mark_bit(current) = marked
         if not atom(current)
            next = left(durrent)
            left(current) = previous
            previous = current
            current = next
      // retreat
      while previous <> null
      and flag_bit(previous) == set
            flag_bit(previous) = unset
            next = right(previous)
            right(previous) = current
            current = previous
            previous = next

      if previous == null
            done = true
      else
      // switch to right subgraph
            flag_bit(previous) = set
            next = left(previous)
            left(previous) = current
            current = right(previous)
            right(previous) = next
```

Algorithmus 6: Pointer Reversal Algorithmus für binäre Knoten, [3]

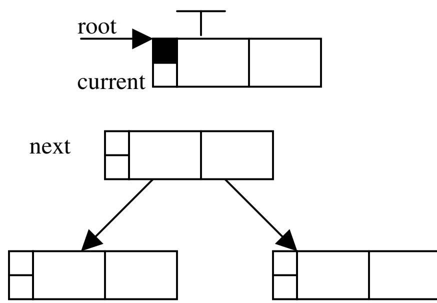
Zur einfacheren Traversierung der Datenstruktur werden 3 Zeiger eingeführt: previous, current und next. Wie die Namen schon verraten, zeigt current auf den gerade besuchten Knoten, previous auf den vorigen, das heißt jenen, der in der Datenstruktur eine Ebene höher ist, und next auf jenen, der von current aus gesehen eine Ebene tiefer ist.

Neu ist hier auch die Existenz eines Flagbits, das eine weitere Fallunterscheidung ermöglicht. Ist es gesetzt, bedeutet das, dass die Adresse des Elternknotens im rechten Feld des current-Knotens zu finden ist. Ist es nicht gesetzt, ist die Adresse im linken Feld zu finden.

Da das Markieren der aktiven Zellen wieder von der Wurzel ausgeht, ist previous am Beginn des Algorithmus nicht definiert. Zuerst wird der linke Ast ganz nach unten verfolgt, also solange, bis man an einem atomaren Knoten angelangt ist oder man auf einen Knoten trifft, dessen Markbit schon gesetzt ist. Auf dem Abwärtsweg wurde wie schon gewohnt das Markbit jedes besuchten Knotens gesetzt. Zusätzlich wurde die Adresse, auf die das linke Feld des Knotens gezeigt hat, in next gespeichert. Dadurch konnte dem linken Feld eine neue Referenz zugewiesen werden, nämlich die Adresse des Elternknotens. Über next war es somit möglich, einen Schritt

in die Tiefe zu gehen, während left den Weg nach oben weist. Zur Veranschaulichung, siehe Abbildung 3:

Davor



Danach

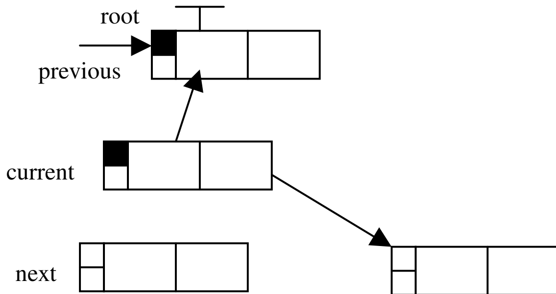


Abbildung 3: Ein Traversierungsschritt nach unten

Beim Weg zurück nach oben kommt jetzt das Flagbit ins Spiel. Ist es gesetzt, so komme ich in den retreat-Zweig des Algorithmus. In meinem Beispiel ist es allerdings nicht gesetzt, sodass jetzt die switch-Phase beginnt. Vom atomaren Knoten bin ich wieder einen Schritt nach oben gewandert, und setze jetzt das Flagbit. Gleichzeitig stelle ich die ursprüngliche Referenz des linken Feldes wieder her, speichere die Adresse des Knotens, zu dem der rechte Zeiger des Knotens führt, in next und lege die Referenz auf den Vaterknoten im reichten Zeiger ab. Anschließend folge ich dem rechten Subgraphen des aktuellen Knotens

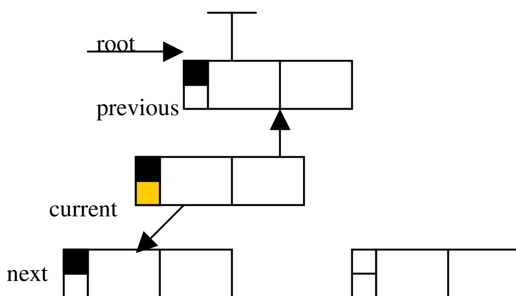


Abbildung 4: Switch

Nach dem Erreichen eines atomaren Knotens beziehungsweise eines Knotens, der schon früher besucht worden ist, am rechten Ast, wird auch hier der Weg nach oben zurückverfolgt. Ich befinde mich nun im retreat-Teil des Algorithmus. Wenn ich nun auf den Verzweigungsknoten treffe, den ich am gesetzten Flagbit erkenne, so lösche ich den Flag, setze die Referenz des rechten Knotens wieder richtig und wandere eine Ebene hinauf.

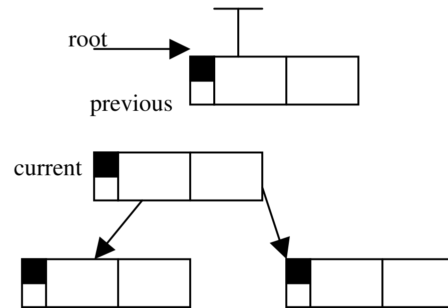


Abbildung 5: Retreat

Dies geschieht solange, bis man wieder am Wurzelknoten angelangt ist (erkennbar, weil previous wieder null ist). Somit ist die gesamte Struktur durchwandert worden, ohne dass es nötig war, einen Verzweigungspunkt im Speicher separat zu vermerken.

Im Gegenzug mussten drei weitere Zeiger mitgeführt werden, außerdem wurde ein zusätzliches (im besten Fall) Bit benötigt. Das schwere Problem eines Stack Überlauf kann bei diesem Verfahren allerdings nicht mehr auftreten.

Der obige Algorithmus ist natürlich auf die Bedürfnisse von Lisp zugeschnitten, wo jeder Knoten nie mehr als zwei Nachfolger hat. Die Erweiterung auf mehrere Zeiger ist jedoch, zumindest von der Idee her, sehr einfach. Statt einem Markierungsbit und einem Flagbit hat man jetzt zwei andere Felder pro Knoten. In einem, ist die Anzahl der Zeiger vermerkt, die vom Knoten wegführen, etwa n. Das zweite Feld, von Thorelli i-Feld genannt, hat zwei Funktionen: als Zähler, wie viele der ausgehenden Referenzen schon abgearbeitet wurden und als Markbit. Zu Anfang wird es auf 0 gesetzt. Bei jedem Besuch des Knotens, was nun natürlich öfter als nur dreimal pro Knoten möglich ist, wird i inkrementiert. Ein i-Wert ungleich Null weist also auf einen aktiven Knoten hin, ein i-Wert von 0 folglich auf einen Knoten, der von Mark nie berührt wurde. [7]

VII. LAZY SWEEPING

Bei allen bisher vorgestellten Mark and Sweep Algorithmen musste der Mutator angehalten werden, während erst Mark und danach Sweep liefen. Diese Nacheinanderausführung ist ein großer Schwachpunkt von Mark and Sweep. Wenig erstaunlich daher, dass ein großes Ziel von Optimierungsbemühungen war, die Zeit zu verringern, die die reguläre Berechnung warten musste. Ein Ansatzpunkt hierfür war die Bereinigungsphase. Da Sweep in den bisherigen Algorithmen den gesamten Heap durchläuft, geht hier viel Zeit verloren. Lazy Sweeping setzt nun mit der

Idee an, Mutator und Sweep gleichzeitig ablaufen zu lassen. Dies ist möglich, da Sweep nur jene Objekte im Speicher verändert, das heißt in die free_list einhängt, die für den Mutator unerreichbar sind.

Anzumerken ist bei diesem Verfahren, dass Mark immer noch beendet sein muss, bevor Sweep an die Reihe kommt. Nachstehende Grafik verdeutlicht den Ablauf:

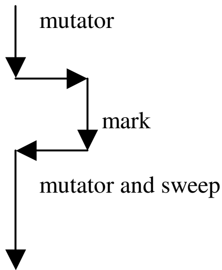


Abbildung 6: Lazy Sweeping [5]

Am einfachsten ist es, bei jeder Allokation dem Sweeper eine bestimmte Zeitspanne zum Bereinigen zu erlauben. Gleichzeitig wird die Bereinigung abgebrochen, sobald Sweep einen Speicherblock im Heap gefunden hat, der groß genug ist, dass der gleichzeitig laufende Mutator ein neues Objekt anlegen kann. In dieser Hinsicht läuft Sweep auch jetzt nur dann, wenn der Mutator freien Speicher benötigt.

Hughes Lazy Sweeping Algorithmus realisiert dieses inkrementelle Sweep, indem er sich in einer globalen Variablen „sweep“ den Platz im Heap merkt, wo Sweep bei der letzen Allokation mit der Bereinigung aufgehört hat. Sweep durchläuft ja immer noch den Heap in linearer Weise, nur dass dies jetzt in vielen kleinen Schritten geschieht. Durch diese Vorgehensweise können neu angelegte Objekte des Mutators (deren Markierungsbit ja nicht gesetzt ist) auch nie zu früh gelöscht werden, da sie ja nur in jenem Teil des Heaps angelegt werden, der von Sweep gerade geprüft wurde. Bevor Sweep wieder zu dieser Speicherzelle kommt, muss ja eine Markierungsphase durchgeführt worden sein, während der das Markbit der Zelle gesetzt wurde. Kommt Sweep am oberen Ende des Heaps an, so ist im Moment kein freier Speicher für den Mutator mehr vorhanden. Letzterer wird somit angehalten und eine neue Markierungsphase wird gestartet. Nach deren Ende beginnt Sweep wieder am Anfang des Heap mit der Bereinigung, während Mutator mit seinen Berechnungen fortfährt.

Bei diesem Verfahren erspart man sich auch das Führen einer Free list, da Speicherzellen sofort an den Mutator zurückgegeben werden, der sie für das Anlegen neuer Objekte verwendet. [3]

VIII. Mark During Sweep

Selbst beim Lazy Sweeping Algorithmus, der die Latenz für den Mutator verringerte, war es nicht möglich, die Markierungsphase und die Sweep-Phase nebenläufig auszuführen. Queinnec, Beaudoing und Queille haben sich mit diesem Thema befasst und den Mark During Sweep Algorithmus erdacht. Hierbei handelt es sich um einen sogenannten parallelen Mark and Sweep Algorithmus.

Bei seinem Entwurf wurde auf die Anforderungen von Embedded Systems Rücksicht genommen. Genauer heißt das, dass Mark During Sweep eine kurze und vorraussagbare Antwortzeit garantiert. Weiters wurde der Algorithmus für Systeme mit begrenztem Speicher optimiert. Um unvorhersehbare Garbage Collection Durchläufe zu vermeiden, wird auch ein garantierter Durchsatz festgelegt. [5]

Da bei Mark During Sweep Mark und Sweep gleichzeitig ablaufen, muss es eine feinkörnige Synchronisation zwischen den beiden Phasen geben. Man stelle sich ein Sweep vor, das aktive Zellen löscht, die nur deshalb weiß markiert sind, weil Mark noch nicht zu ihnen vorgedrungen ist. Zur Vermeidung solcher Fehler ist es notwendig, nun neben den beiden schon bekannten Zuständen einer Speicherzelle (aktiv oder schwarz, nicht lebendig oder weiß) eine weitere Farbe einzuführen: grau. Eine graue Markierung bedeutet: möglicherweise ist diese Zelle schwarz, der Marker ist aber noch nicht zu dieser Zelle vorgedrungen und eine endgültige Entscheidung steht somit noch aus. Sweep löscht nur weiße Zellen, das Löschen einer grauen Zelle ist für Sweep verboten. Am Ende der Markierungsphase sind daher keine grauen Speicherzellen mehr vorhanden, da ja zu diesem Zeitpunkt ohne Zweifel feststeht, welche Zellen garbage und welche noch aktiv sind.

Es gibt noch einen zweiten Grund, warum graue Zellen benötigt werden, und zwar um die Invariante des Mark During Sweep Algorithmus aufrecht zu erhalten. Diese lautet: Keine schwarze Zelle darf eine Referenz auf eine weiße Zelle enthalten. Kommt es zu solch einer Situation, so wird einer der beiden Knoten grau eingefärbt, und die Invariante wurde nicht verletzt.

Da der Sweeper immer noch auf die Markierungen des Markers angewiesen ist, benutzt Mark During Sweep einen Trick, um beide gleichzeitig ausführen zu können: es werden immer 2 Generationen zugleich betrachtet. Ist die Markierung einer Generation beendet, kann das Bereinigen dieser Generation beginnen. Zur gleichen Zeit beginnt die Markierung der nächsten Generation. In einer Grafik sieht dies folgendermaßen aus:

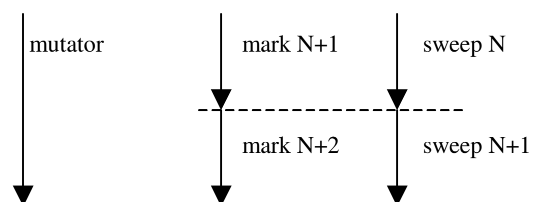


Abbildung 6: Mark During Sweep [5]

Jede der beiden Generationen hat hierbei ihre eigene Farbe, so dass zu jeder Speicherzelle eine Colour N sowie eine Colour N+1 gespeichert werden muss.

Zusätzlich gilt noch, dass das Umfärben einer Zelle in einer atomaren Operation erfolgt.

Im folgenden werde ich den Algorithmus nur verbal beschreiben, eine Definition in Pseudo-Code ist in [5] zu finden. Die Autoren sprechen auch davon, dass sie an einer Implementierung ihres Algorithmus arbeiten, daher liegen keine Daten über die Performanz von Mark During Sweep vor.

A. Mark

Der Marker operiert auf einer bestimmten Generation, nehmen wir an N+1. Zu Beginn der Markierungsphase sind, wie es auch bei den bisherigen Algorithmen der Fall war, keine Markierungsbits gesetzt. Das heißt, alle Speicherzellen haben Colour N+1 grau oder weiß. Zu Beginn werden alle Wurzeln grau gefärbt. Im Unterschied zu den bisherigen Definitionen von Mark, folgt die Markierungsphase von Mark During Sweep nicht den Referenzen, die von den Wurzeln oder aktiven Zellen wegführen. Im Gegenteil, es wird der Heap solange linear durchsucht, bis keine einzige grauen Zelle mehr im Heap gefunden wird.

Hat eine Zelle Colour N+1 gleich schwarz (schon markiert) oder weiß ((noch) nicht markiert), so wird mit der nächsten Zelle weitergemacht.

Trifft Mark auf eine graue Zelle, wird diese schwarz gefärbt, alle Kinder dieser Zelle werden grau gefärbt. Der Zähler, der das lineare Durchlaufen des Heaps ermöglicht, wird zurückgesetzt und der Heap wird erneut von seinem unteren Ende weg gescannt. Dies ist deshalb notwendig, da man ja nicht wissen kann, ob ein eben grau gefärbtes Kind nicht „vor“ dem schwarzen Vaterknoten im Heap gespeichert ist und von Mark bisher nicht beachtet wurde, weil es weiß war. Trotz dieses Resets terminiert Mark jedoch immer, weil es irgendwann keine Zellen mit Colour N+1 gleich grau mehr geben wird, da diese alle schwarz gefärbt wurden.

B. Sweep

Der Sweeper, der zeitgleich mit obigem Marker läuft, operiert auf der vorherigen Generation, also auf Colour N. Er hinkt somit eine Generation hinter dem Marker und dem Mutator hinterher. An seinem Beginn gibt es keine grauen Zellen mehr (diese wurden ja von Mark gefärbt). Außerdem steht fest, dass alle Zellen, deren Colour N weiß ist, garbage sind. Der Heap wird wieder von einem Ende zum anderen durchwandert.

Trifft Sweep auf eine weiße Zelle, wird diese in die free_list eingehängt. In diesem Fall wird die Colour N+1 dieser Zelle grau gesetzt. Dies wird durch das Einhalten der Invariante bedingt. Mark operiert ja in diesem Moment auf Generation N+1 und hat deren Wurzeln erst grau, danach schwarz gefärbt. Unter den Wurzeln ist selbstverständlich auch der Beginn der free_list. Ohne Umfärben würde ein schwarzer free_list Knoten auf den weißen, eben freigegebenen Garbageknoten zeigen.

Trifft Sweep auf eine schwarze Zelle, wird deren Colour N auf weiß zurückgesetzt.

Somit haben am Ende der Bereinigungsphase alle Zellen Colour N gleich weiß und der Speicher ist bereinigt worden.

C. Kollektor

Der Kollektor stößt Mark und Sweep an. Er achtet ebenso auf die Synchronisation zwischen diesen beiden Phasen und darauf, dass am Ende eines Mark/Sweep Zyklus die Generationennummer erhöht wird.

Da der Mutator und der Marker auf derselben Generation operieren, und somit beide die Farbe der aktuellen Generation verändern können, ist die Synchronisation von Mark During Sweep alles andere als trivial.

IX. Very Concurrent Garbage Collection

Ein Algorithmus, der diese komplizierte Synchronisation entschärft, nennt sich Very Concurrent Garbage Collection, im folgenden nur noch mit VCGC bezeichnet. In Pseudocode sieht er folgendermaßen aus:

```
big int epoch = 2;
root_set_t roots = {};
thread_t mutator, marker, sweeper;

forever{
    mutator <- make_thread mutate(Color(epoch));
    marker <- make_thread mark(Color(epoch), roots);
    sweeper <- make_thread sweep(Color(epoch-2));
    barrier_sync {marker, sweeper}
   //invariant: all reachable data have Color(epoch)
    suspend_thread(mutator);
    roots <- get_roots(mutator);
    delete_threads{mutator, marker, sweeper}
    epoch++;
}
```

Algorithmus 7: Very Concurrent Garbage Collection, [2]

VCGC verwendet 3 nebeneinanderlaufende Threads. Die Farben schwarz, grau und weiß von Mark During Sweep werden hier durch die Farben drei unterschiedlicher Epochen verstanden. Jede Epoche hat hierbei eine eindeutige Farbe. Epochen werden modulo 3 berechnet, so dass 3 unterschiedliche Farben ausreichend sind. Die Barriere stellt sicher, dass sowohl der Sweeper als auch der Marker ihre Arbeit beendet haben, bevor eine neue Epoche angestoßen wird. Wie funktioniert nun VCGC?

A. Mutator

Der Mutator wird mit der Farbe der aktuellen Epoche initialisiert, der sogenannten Mutatorfarbe. Der zur Allokation zur Verfügung stehende Speicherbereich ist in VCGC wieder als Liste implementiert. Der Mutator sucht in der Liste nach einem geeigneten Speicherblock, beim anschließenden Belegen der Speichezelle wird diese mit der Mutatorfarbe markiert. Somit ist sichergestellt, dass die soeben angelegte Zelle für mindestens zwei Epochen nicht zu garbage wird, da der Sweeper ja zwei Epochen hinter dem Mutator arbeitet.

Da sich der Umgang des Mutators mit der Farbe einer Speicherzelle auf das Markieren beim Allozieren beschränkt, kommt es zu keinen race conditions mit dem Marker oder dem Sweeper.

Einen Engpass zwischen Mutator und Sweeper stellt die free_list dar. Feinkörnige Synchronisation kann hier aber

vermieden werden, wenn jeder Thread einen eigenen Zeiger auf die free_list hat. Der Sweeper fügt Elemente hinzu, der Mutator entfernt Elemente; somit hat man ein Producer-Consumer Problem, für das es erprobte Synchronisationsmechanismen gibt. [2]

B. Marker

Der Marker wird ebenfalls mit der aktuellen Farbe initialisiert. Zusätzlich erhält er noch einen Verweis auf die Wurzeln, da VCGC wieder dem klassischen Verhalten eines Markers entspricht: von den Wurzeln ausgehend aktive Zellen aufzuspüren.

Jede Zelle, die der Marker antrifft, kann entweder Color(epoch) haben oder Color(epoch-1). Im ersten Fall handelt es sich entweder um eine vom Mutator neu angelegte Zelle oder um eine Zelle, die der Marker schon über eine andere Referenz besucht hat; folglich muss der Marker keine weitere Aktion setzen. Im zweiten Fall handelt es sich um eine Zelle, die der Marker im vorherigen Durchlauf markiert hat (daher Color(epoch-1)), die er aber im aktuellen Durchlauf noch nicht angetroffen hat. Um die Zelle auf den neuesten Stand zu bringen, markiert er sie also mit der aktuellen Mutatorfarbe. Wenn der Marker die Synchronisationsbarriere erreicht, ist garantiert, dass alle aktiven Speicherzellen die aktuelle Epochenfarbe haben.

Der Marker ist der einzige Thread, der die Farbe einer Zelle ändern kann. Der Mutator setzt sie lediglich bei der Allokation, der Sweeper prüft Farben nur, schreibt sie aber nie. Dadurch ist keine feinkörnige Synchronisation notwendig. [2]

C. Sweeper

Der Sweeper wird mit Color(epoch-2), der sogenannten Sweeperfarbe, initialisiert; er hinkt den beiden anderen Threads also 2 Epochen hinterher. Nur jene Zellen werden demnach gelöscht, deren Farbe zwei Epochen hinter der aktuellen Mutatorfarbe liegt.

Beim sequentiellen Durchwandern des Heaps prüft der Sweeper die Farbe eines jeden Speicherobjekts. Objekte mit Sweeperfarbe werden an die free_list angehängt. Objekte mit anderer Farbe werden vom Sweeper nicht verändert.

Der Sweeper kann somit nie vom Mutator in diesem Durchlauf angelegte Zellen freigeben, das diese ja Color(epoch) haben. Mit dem Marker gibt es ebenfalls keine Konflikte, da alle aktiven Zellen Color(epoch) oder aber Color(epoch-1) haben. Somit ist es nicht möglich, dass Sweeper Daten wegwirft, die vom Mutator noch benötigt werden.

Die Barriere ist der einzige Synchronisationspunkt im Algorithmus, die aber nur sehr selten auftritt. Sie ist notwendig, da sichergestellt werden muss, dass der Marker alle aktiven Objekte mit der aktuellen Epochenfarbe aktualisiert hat, bevor der Sweeper der nächsten Epoche startet.

Zwar wird auch bei VCGC der Mutator angehalten, jedoch nur für jene Zeit, die das System benötigt, um das aktuelle Wurzelset abzuspeichern, das ja in der neuen Epoche an den neu angelegten Marker übergeben werden muss. Die aktuellen Threads werden gelöscht, eine neue Epoche berechnet und die neuen Threads gestartet.

D. Implementierungen

Die Autoren haben ihren Algorithmus auf zwei sehr unterschiedlichen Systemen implementiert. Zum einen auf dem kommerziellen Inferno Betriebsystem, mit dem Hauptzweck zyklische Datenstrukturen zu erkennen und zu löschen. Zum anderen im SML/NL ML Compiler, wo VCGC lange Garbage Collection Pausen verhindern kann, während er gleichzeitig den Speicherverbrauch reduziert. [2]

X. Zusammenfassung

Ich habe einen Überblick über die in der Literatur beschriebenen Garbage Collection Algorithmen gegeben, die nach dem Mark and Sweep Prinzip arbeiten. Auf konkrete Implementierungen, Performanztests und ähnliches bin ich nicht näher eingegangen. Da es sich um Algorithmen aus drei Jahrzehnten handelt, wären etwaige Messergebnisse nicht miteinander vergleichbar, beziehungsweise würde ein solcher Vergleich nicht aussagekräftig sein.

XI. Literatur

[1] [Boehm88] Boehm H.J., Weiser M.: 1988. Garbage collection in an uncooperative environment. Software -- Practice and Experience. 18(9):807-820

[2] [Huelsbergen98] Lorenz Huelsbergen and Phil Winterbottom. Very concurrent mark-&-sweep garbage collection without fine-grain synchronization. Symposium on Memory Management, Vancouver, October 1998. ACM Press, pages 166-175

[3] [Jones96] Jones, R, Lins, R.: Garbage Collection: Algorithms for Automatic Dynamic Memory Management. John Wiley 1996

[4] [Knuth73] Donald E. Knuth. *The Art of Computer Programming*, volume I: Fundamental Algorithms, chapter 2. Addison-Wesley, second edition, 1973.

[5] [Queinnec89] Christian Queinnec, Barbara Beaudoing, and Jean-Pierre Queille. Mark DURING Sweep rather than Mark THEN Sweep. Lecture Notes in Computer Science, 365:224-237, 1989

[6] [Schorr67] Schorr, Waite: An efficient machine-independent procedure for garbage collection in various list structures. Communications of the ACM (August 1967), 501-505

[7] [Thorelli72] Lars-Erik Thorelli. Marking algorithms. *BIT*, 12(4):555-568, 1972.

[8] [Zorn90] Benjamin Zorn. Comparing Mark-and-sweep and Stop-and-copy Garbage Collection. ACM conference on LISP and functional programming, 1990