Minneshantering i högnivåspråk

Björn Mosten

December 3, 2019

1 Inledning

Varje program använder sig utav information, eller "data" - som är ett mycket mer passande begrepp med tanke på ämnet som avhandlas i den här essän. Data används, manipuleras och sparas men majoriteten av datan i de flesta program används bara temporärt och förlorar sin relevans så fort den inte längre kommer användas utav någon annan del av programmet. Det är möjligt att bara låta datan vara, men en dators minne är ändligt och det är således alltid önskvärt att göra sig av med den data som inte längre fyller någon som helst funktion. Detta leder oss in på ämnet *minneshantering*.

2 C

2.1 Minneshantering i C

2.1.1 Manuell allokering

Vid utveckling av program i C måste minne allokeras manuellt till heap om datan ska vara tillgänglig för andra delar av programmet. Vid allokering måste också programmet veta exakt hur mycket minne som skall allokeras och detta är återigen programmerarens uppgift att förse programmet med. Oftast är det så enkelt som att köra funktionen *sizeof* för att utröna det minne som krävs, men det händer att processen blir mer avancerad än så om datatyperna man hanterar är mer invecklade.
Programmeraren har således all kontroll, och den kontrollen sträcker sig dessutom till frigöring av detta allokerade minne. Minne som är allokerat är just det - allokerat; upptaget. Inget annat program och ingen annan del av programmet har tillåtelse att fördela sin data till, eller allokera, till den platsen i minnet.

2.1.2 Manuell frigöring

Programmering i C kräver inte enbart att allokering av minne sker manuellt, utan också att frigörandet av det allokerade minnet utförs manuellt - åtminstone om man vill skriva ett korrekt program med bra kod utan minnesläckor. Detta kan vara komplicerat, speciellt för oerfarna programmerare då det inte alltid är uppenbart vilket minne som inte längre är relevant. Om minne frigörs för tidigt kan det leda till diverse buggar, ofta segmentation faults, men också helt felaktiga värden i den händelse att en annan del av programmet lagt in ett annat värde på den adressen i minnet.

3 Java

3.1 Minneshantering i Java

3.1.1 Minnesallokering i Java

Minnesallokering i Java sker automatiskt på så sätt att minne allokeras utan att programmeraren behöver förbereda något innan ett objekt skapas och placeras på heap. Programmeraren har en viss kontroll över *när* minne allokeras, nämligen då programmeraren skapar ett nytt objekt genom new-operatorn. På grund av att adresser (e.g. referenser till objekt) i Java ej kan manipuleras så kan inte referenser peka mot ett felaktigt objekt (vilket är möjligt i C).

3.1.2 Frigöring i Java

Java är ett mer abstrakt språk än vad C är och många funktioner och system är gömda från programmeraren. Det finns inget sätt för en Java-programmerare att manuellt frigöra minne - frigöring sker genom något som kallas för Garbage Collection (GC).

4 Gargabe Collection och olika varianter.

GCs funktion är i princip att frigöra alla objekt som saknar referenser och därav är irrelevanta för programmet. Detta kan ses en otrolig graciös funktionalitet för nya Java-programmerare då man endast behöver nullifiera referenser till de objekt som inte längre behövs. Garbage Collection är dessutom fördelaktigt för oerfarna programmerare eftersom det betydligt reducerar risken för minnesläckor.

Det finns dock nackdelar med hela konceptet av GC i Java. Garbage Collection sker endast när systemet anser att det är säkert att göra det vilket innebär att, även om vissa objekt saknar referenser, kan det ta tid innan de referenslösa objekten faktiskt frigörs, vilket inte är optimalt. Man kan explicit be Java att utföra en Garbage Collection men systemet är producerat på så sätt att det, trots det explicita kallet, kan ta tid innan GC sker om det ens sker överhuvudtaget.

4.1 Mark and Sweep

Mark and Sweep-algoritmen för GC består av två faser; mark (markeringsfasen) och sweep (svepfasen) [2]. Alla objekt klassificeras preliminärt som 'döda' när de skapas. GC skapar träd som utgår ifrån bland annat lokala variabler, funktioner som körs, aktiva trådar och statiska fält från laddade klasser. Vid det här laget består träden bara av rötter, så GC följer referenser från rötterna till andra objekt och varje objekt som GC söter på markeras som levande.
När markeringsfasen är över börjar svepfasen och alla objekt som fortfarande är klassificerade som döda kan frigöras.
Nackdelen med Mark and Sweep är att programmet temporärt måste stannas upp för att GC ska kunna få arbeta. Detta beror på att en hel del problem kan uppstå när man bygger upp ett träd av data som kontinuerligt förändras [1].

4.2 Reference Counting

Reference counting sker genom att programmet håller koll på hur många referenser som pekar till varje Java-objekt. När antalet referenser till ett objekt blir 0 innebär det att objektet är

irrelevant och minnet kan frigöras omedelbart - till skillnad från Mark and Sweep körs inte Reference Counting i bakgrunden och objekt raderas i realtid [3].
Eftersom varje objekt kräver en tillhörande räknare så krävs också mer minne för att husera dessa räknares värden. Det finns också en risk för ofrigörliga objekt att uppstå när Reference Counting används som GC. Det kan inträffa när två objekt refererar varandra och inget annat objekt refererar till något av de två objekten (vilket innebär att båda objekten egentligen är irrelevanta och bör tas bort) så kommer de aldrig raderas eftersom deras reference count alltid kommer vara större än 0. Detta går att komma runt men enbart med resurskrävande analyser.

4.3 Jämförelse

Båda GC-metoder som presenterats i detta dokument har både fördelar och nackdelar vilka kan vara mer eller mindre betydande beroende på vad det är man försöker åstadkomma.

4.3.1 Throughput

Throughput (här används det engelska ordet då det saknas en bra svensk motsvarighet) är ett mått på hur snabbt information kan behandlas. Generellt sett tar ett program emot data, behandlar den och producerar sedan någon sorts output. I de flesta fall är det önskvärt att ha så hög throughput som möjligt. I somliga fall kan en hög throughput krävas för att ett system skall fungera korrekt.
Eftersom Reference Counting kräver en ökning av ett värde vid alla operationer som begär det innebär det att throughput kommer vara lägre men relativt konstant.
Mark and Sweep kommer däremot erbjuda en högre peak throughput men med intervaller av lägre throughput när GC genomförs. Vilken av metoderna som medför högst throughput kommer bero på varderas implementation och programmets natur.

4.3.2 Latency and jittering

Latency är den period av tid då GC aktiverats och programmet pausat exekvering av samtliga trådar[4]. Latency är i den här kontexten enbart ett fenomen hos Mark and Sweep och inte hos Reference Counting eftersom Reference Counting aldrig stoppar exekvering utav någon del av ett program.
I de fall där latency kan innebära problem bör således Reference Counting användas som GC. Flimmer (jittering) kan vara ett problem för program som behöver utföra handlingar vid specifika tidpunkter eller under fasta intervaller. Reference Counting medför lågt konstant flimmer då counters ökar och objekt frigörs "on-the-fly". Mark and Sweep medför stort flimmer men under mycket korta perioder. Återigen, vilket system som är bäst beror på vad det är som ska skapas.

5 Avslutande ord

Det finns för- och nackdelar med alla metoder gällande memory management och GC. Det är (som med många andra saker) upp till utvecklaren att lista up och besluta om vilken som bäst passar dennes projekt bäst.

References

[1] Andreasson, Eva (2019): *JVM Performance Optimization*
.

[2] Author, Unknown (2019): *GC Algorithms: Basics*
.

[3] Sridharan, Mohanesh (2019): *Garbage Collection vs Automatic Reference Counting*
.

[4] Wenzel, Maira (2019): *Latency Modes*
.