Obligatorisk opgave 3

Operativsystemer og C

Bachelor in Software Development,
IT-University of Copenhagen

Jakob Melnyk, jmel@itu.dk
Frederik Lysgaard, frly@itu.dk
Jacob Claudius Grooss, jcgr@itu.dk

November 30 th, 2012

Contents

1 Forord

I denne rapport dokumenterer vi vores valg i forhold til implementationen af opgaverne i Obligatorisk opgave 3.

Når vi refererer til "Programming Language Concepts", mener vi "Peter Sestoft, Programming Language Concepts, ISBN 978-1-4471-4156-3 (eBook)".

Vi har testet vores tre implementationer af garbage collection algoritmer på eksemplerne "ex30.lc", "ex35.lc" og "ex36.lc", som beskrevet i opgaven. Disse fulgte med "listc.zip"[1] mappen, som blev givet med opgaven.

Kildekode og testdokumentation kan findes i appendix på side 6. Vores git repository kan findes på https://github.com/esfdk/BOSC/tree/master/oo3.

[1] https://blog.itu.dk/BOSC-E2012/files/2012/11/listc.zip

2 Beskrivelse af implementation

2.1 Opgave 1

2.1.1 Del 1

ADD untagger de to øverste elementer på stakken, ligger dem sammen, tagger den nye værdi og ligger den på toppen af stakken.

CSTI I tager den næste værdi i p[ ] arrayet og ligger den på toppen af stakken.

NIL Ligger 0 på toppen af stakken. Hvis der kun ligger nul bits betyder det NIL og ikke integer nul.

IFZERO tager det øverste element af stakken og decrementere stackpointeren med en. Den tjekker om v er en int. Hvis v er en int udtagges v og sammenlignes med nul , ellers sammenlignes v med NIL. Hvis sammenligning er sand i tilfældet med nul bliver program counter sat til den nuværende værdi på p[pc] ellers bliver næste instruktion udført.

CONS laver en cons celle ud af de to øverste elementer på stakken, og decrementere så stack pointeren med en.

CAR henter et word fra stakken og tjekker om det er NIL, hvis det ikke er NIL tages det første element af cons cellen og ligges på toppen af stakken i stedet for det hentede word.

SETCAR henter det øverste element på stakken og et word. Den sætter wordets første værdi til at være den udhentede værdi på stakken.

2.1.2 Del 2

Length laver to right shifts, hvilket fjerner de to garbage collection bites. Derefter bruger den bitwise AND til at sammenligne length bitsne med 0x003FFFF. Dette giver os værdien af n bitsne.

Color går ind og bruger bitwise AND til at sammenligne farven på cellen med 0011 hvorved den finder cellens faktiske farve.

Paint går ind og ændrer gg til den color der er blevet givet med som argument. eksempelvis laver Paint med arguemnt BLUE gg om til 11.

2.1.3 Del 3

allocate( ) bliver kun kaldet i CONS casen. (umiddelbart ikke andre stedder end i interpretation loopet)

2.1.4 Del 4

Collect() bliver kaldet når der bliver allokeret og der ikke er noget free space.

2.2 Opgave 2, 3 og 4

Igennem opgaverne har vi skulle lave tre forskellige implementationer af garbage collection. En mark-sweep med rekursive funktions kald, en mark-sweep uden rekursive funktions kald og en stop-and-copy med rekursive funktions kald. For at undgå at en fil indeholdte flere implementationer med nogle af dem udkommenteret, valgte vi at dele dem ud i flere filer. Opgave 2, 3 og 4 findes i listmachineV1.c, opgave 5 i listmachineV2.c og opgave 6 i listmachineV3.c.

Vi har valgt at beskrive vores implementation af opgave 2, 3 og 4 i samme afsnit, da 3 og 4 går ud på at forbedre algoritmen vi implementerede i opgave 2.

Vores implementation of mark-sweep algoritmen består af tre dele; en mark()-funktion, en markPhase()-funktion og en sweepPhase()-funktion.

2.2.1 Opgave 2

I opgave 2 implementerede vi vores første udgave af en simpel mark-sweep garbage collector. Som indholdte mark(word*block), markPhase(int s[], intsp) og sweepPhase(). 'mark(word* block)'-funktionen bruges til at farve hvide blokke sorte. Funktionen tjekker først om blokken er farvet (linje 412-416). Er den det, retunerer funktionen, da der ikke skal ske mere. Er den ikke farvet, bliver den farvet sort, hvorefter hvert word der kan nås fra blokken bliver gennemløbet. Hvis wordet ikke er en int og ikke er nil, kaldes mark()-funktionen på wordet (linje 419-428).

'markPhase(int s[], int sp)'-funktionen starter markeringen af blokke. Funktionen looper igennem alle elementer på stacken, og hvis elementet ikke er en int og ikke er nil, castes det til et word og 'mark()' kaldes med wordet some parameter. Disse to funktioner udgør tilsammen mark fasen af algoritmen.

sweepPhase() går igennem heapen og putter alle hvide bloks over i freelisten(linje 485-487) og maler sorte blocks hvide(linje 491-493).

2.2.2 Opgave 3

I opgave 3 bliver vi bedt om at forbedre sweepPhase() så den kan slå tilstødende hvide blokke sammen. Det gør vi ved at tjekke om den næste blok er hvid og den næsten blok stadig er i heapen. Mens det er sandt øge vi længden af "free space" og sætter block headeren til en junk value(linje 462-471).

2.2.3 Opgave 4

'sweepPhase()'-funktionen er den funktion, som fjerner alt det fundne garbage (linje 445-509). Den looper igennem hele heapen, og tjekker hvilken farve de forskellige words har. Hvis de er grå eller blå, sker der ingenting. Hvis de er sorte bliver de farvet hvide. Hvis de er hvide sker følgende:

1. Det næste word i heapen bliver læst. (linje 459)
2. Der tjekkes om det næste words farve er hvid og om (linje 462)
3. Hvis tjekket bestås, fortæller vi programmet, at der skal afsættes ekstra plads, så vi kan slå blokkene sammen. Derefter indlæses det næste word igen. (linje 464-470)
4. Det tjekkes om der skal afsættes ekstra plads til at slå flere blokke sammen. Hvis der skal, bliver stedet i heapen sat til resultatet af et kald til 'mkheader' macroen med det originale words tag, længden af det originale word plus den ekstra plads og med farven blå. Hvis ikke, bliver stedet i heapen bare farvet blå. (linje 473-482)
5. Til slut bliver wordet lagt over i freelist. (linje 485-487)

Måden vi har implementeret 'sweepPhase()' på, gør at den kan slå flere døde blokke sammen til en død blok, hvorved vi undgår fragmentation.

2.3 Opgave 5

For at implementere den ikke-rekursive version af Mark and Sweep[1], har vi fjernet vores "mark" funktion og udvidet "markPhase" funktionen. "sweepPhase" er den samme som i 2.2.

Vores kode kan findes i "oo3/listmachineV2.c" samt på side **??** i appendix. "markPhase" funktionen er på linje 409-453 og "sweepPhase" funktionen er på linje 456-521.

Step A Mal alle blokke, som kan nås fra stakken, grå.

Step B Løb heap igennem og marker alle grå blokke sort.

Step B.1 Hvis en grå blok pegede på en hvid blok, males den hvide blok grå. Hvis der er grå blokke i heap, gentag step B, ellers gå til step C.

Step C Sweep på samme måde som i opgave 2.

2.4 Opgave 6

Vi har implementeret algoritmen til opgave 6 (two-space stop and copy) som en række steps. Algoritmen er forklaret i "Programming Language Concepts"[2]. Vores implementation kan findes i "oo3/listmachineV3.c" samt på side **??** i appendix.

Vi har delt algoritmen op i følgende steps:

Step A Peg freelisten på starten af heapTo.

Step B Iteration over stacken for at finde heap referencer.

Step B.1 Undersøg om blokken er flyttet allerede.

Step B.2 Lav en ny blok og flyt freelist pointeren.

Step B.3 Kør blokken igennem og kopier ord fra blokken.

Step B.4 Returner pointer til den nye blok i heapTo.

Step C Iteration over heapTo for at finde eventuelle referencer til heapFrom.

Step D Ombytning af heapFrom og heapTo.

De to steps er delt ud på to funktioner, som foreslået i opgavebeskrivelsen. "void copyFromTo(int s[], int sp)" (linje 418-461) og "word* copy(word* oldBlock)" (linje 379-415).

Vi har valgt kun at beskrive step B, B.1, B.3 og C i detaljer, da vi føler, at de andre er beskrevet grundigt nok i listen over steps.

2.4.1 Step B

Step B er algoritmens mest komplekse step. Derfor har vi delt det op i fire mindre steps.

Step B løber stacken igennem og undersøger, om den indeholder nogen heap referencer. Hvis der er, bliver de kopieret og stacken bliver opdateret med addresserne på de kopierede blokke.

[1] "Programming Language Concepts", side 192-193
[2] "Programming Language Concepts", s. 181-182

Step B.1 Hvis en blok allerede er kopieret, returnerer vi bare "to-heap" addressen på blokken.
Vi ved, at en blok er kopieret, hvis det holder sandt at "(oldBlock[1] != 0 && !IsInt(oldBlock[1]) && inToHeap((word*) oldBlock[1]))".
Vi bruger det første ord efter headeren i "from-heap" til at holde en "forwarding pointer" til blokken i "to-heap".

Step B.3 Step B.3 løber ordene i blokken fra "from-heap" igennem og kopierer dem til "to-heap".
Hvis et ord er en heap reference, bliver "copy()" kaldt rekursivt, så vi kan få referencen til den blokken i "to-heap".
Hvis det ikke er en heap reference, bliver indholdet af ordet bare kopieret.
Vi kontrollerer desuden om det første ord efter headeren er kopieret - hvis den er, så sætter vi "forwarding pointeren" i "from-heap"s "oldBlock[1]" til addressen på den nyligt kopierede blok.

Step C I step C løber vi heap'en igennem og ser, om der er blokke, som indeholder ord, der peger i "from-heap". Hvis der er, så opdaterer vi referencen ved at hente forwarding pointeren.

A Test

A.1 Opgave 1

A.2 Opgave 2

A.3 Opgave 3

A.4 Opgave 4

A.5 Opgave 5

A.6 Opgave 6

B Kode

B.1 Opgave 1

B.2 Opgave 2

B.3 Opgave 3

B.4 Opgave 4

B.5 Opgave 5

B.6 Opgave 6