24/6/2022

COPYING GARBAGE COLLECTOR IN C

HY-446

ΚΥΡΙΑΚΗ ΜΗΝΙΑΔΟΥ & ΑΙΚΑΤΕΡΙΝΗ ΜΑΡΙΑ ΓΕΡΑΚΙΑΝΑΚΗ
CSD4220 & CSDP1263
csd4220@csd.uoc.gr & agerakianaki@csd.uoc.gr

Περιεχόμενα

Copying Garbage Collector

Κατά την διάρκεια έρευνας για την επιλογή project στο μάθημα, ο Copying Garbage Collector μας κίνησε το ενδιαφέρον. Καθώς δεν καταλαβαίναμε πως λειτουργεί ακριβώς και η αλγοριθμική του υπόσταση δεν μας ήταν ξεκάθαρη, ένα από τα αναπόσπαστα κομμάτια της παρούσας εργασίας είναι η κατανόηση του.

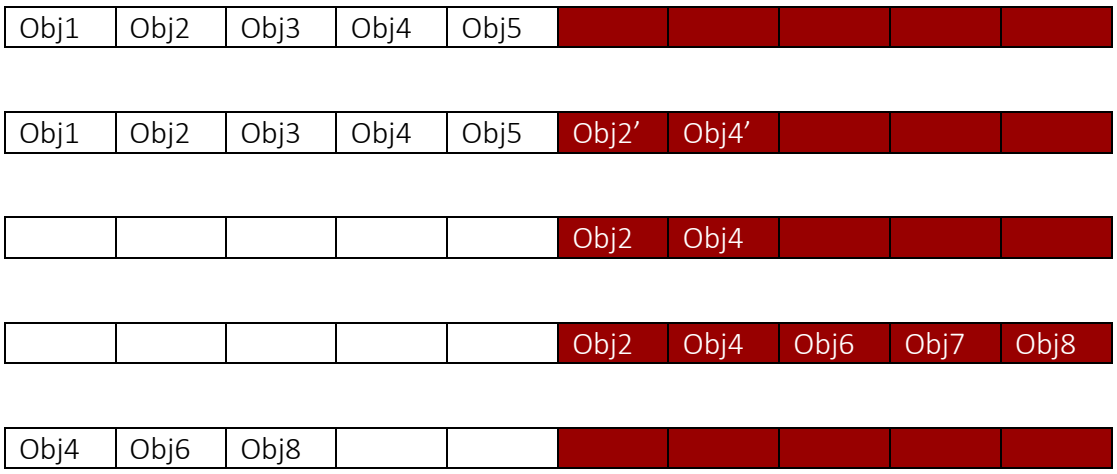
Αρχικά έπρεπε να κατανοήσουμε τον βασικό στόχο των garbage collectors. Κυρίως στόχος τους είναι να ανακαλύπτουν με αυτόματο τρόπο και να ανακτούν τα τμήματα της μνήμης που χρησιμοποιούνται σε κάθε «computation». Οι garbage collectors, στην πλειοψηφία τους βασίζονται στην ιδέα «reclaiming whole object» όταν αυτά δεν είναι πλέον προσιτά από το root set.

Πιο συγκεκριμένα, ο Copying Collector πρακτικά ξεκινά από ένα set of roots και κάνει διάσχιση σε όλα τα προσιτά memory-allocated αντικείμενα και τα αντιγράφει από το ένα μισό της μνήμης στο άλλο.

Η μνήμη από την οποία αντιγράφονται τα αντικείμενα ονομάζεται old space ενώ η μνήμη στην οποία τα αντιγράφουμε ονομάζεται new space. Η αντιγραφή των «reachable» αντικειμένων και η συμπύκνωση τους σε συνεχόμενο κομμάτι γίνεται με στόχο να μην υπάρχουν δεδομένα που δεν είναι χρήσιμα.

Τέλος, στόχος είναι να καταλήξουμε σε ένα συμπιεσμένο αντίγραφο δεδομένων με νέα δεδομένα τα οποία να δίνουν μια συνεχόμενη και μεγάλη περιοχή της μνήμης. Ενώ οι ρόλοι του old και του new space αλλάζουν σε κάθε garbage collection.

Παρακάτω υπάρχει ένα πολύ απλό παράδειγμα για το πως λειτουργεί ο Copying Garbage Collector:



Copying Garbage Collector In C

Τι σκοπιμότητα όμως έχει η υλοποίηση του Copying Garbage Collector στην γλώσσα C; Η βασικότερη απορία μας ήταν γιατί η C, μια υπερβολικά γνωστή και χρήσιμη γλώσσα δεν είχε Garbage Collector.

Μετά από εκτενή έρευνα ανακαλύψαμε ότι η κουλτούρα της C γενικά φαίνεται να στηρίζει την φιλοσοφία ότι το storage management είναι ευθύνη του προγραμματιστή. Αν και αυτή η ευθύνη είναι συναρπαστική για όλους μας, δεν είναι απόλυτα κατανοητό γιατί δεν υπάρχει ο garbage collector στην C και κατά συνέπεια γιατί να μην επιλέγει ο προγραμματιστής αν θέλει αυτή την ευθύνη ή προτιμάει να «δώσει» την ευθύνη αυτή στον garbage collector.

Η υλοποίηση του Garbage Collector στην C φαίνεται να έχει πολλές τεχνικές δυσκολίες. Για παράδειγμα είναι ακριβό να πραγματοποιηθεί λόγω του ότι η δημιουργία του συνεπάγεται και με την δημιουργία άλλων πραγμάτων που κάνουν τον language implementation της C αργό.

Αρχικοί στόχοι

Σε όλα τα project ορίζονται κάποιοι αρχικοί στόχοι. Σε αυτή την παράγραφο αναλύονται οι στόχοι που καθορίσαμε όταν αρχίσαμε να μελετάμε το τι χρειάζεται η υλοποίηση του Copying Garbage Collector.

- Τα συνεχόμενα system calls είναι ακριβά, συνεπώς θα ζητάμε ένα προκαθορισμένο μέγεθος με system call.
- Η αντίστοιχη malloc() που θα χρειαστεί να γράψουμε θα είναι first fit.
- Ο Garbage Collector αλγόριθμος θα ξεκινά από ένα set root (initialized data, BSS, stack).
- Θα χρειαστεί να γραφεί assembly σε κάποια σημεία.
- Ο αλγόριθμος θα είναι Two-Space Collection.

Προβλήματα κατά την υλοποίηση

Φυσικά, το project είναι challenging και αν και δεν καταφέραμε να το υλοποιήσουμε πλήρως (και όπως θα θέλαμε) μας προσέφερε πολλά εφόδια και γνώσεις η ενασχόληση με αυτό.

Συνεπώς σε αυτή την παράγραφό παραθέτουμε όσα μάθαμε ή/και όσα πράγματα δεν έχουμε καταλάβει πλήρως πως λειτουργούν αλλά ήρθαμε αντιμέτωπες με αυτά.

- Τα C προγράμματα σε C χρειάζονται έναν linker με την τελική τοποθεσία των root στην μνήμη.
- Τα Unix συστήματα παρέχουν linkers που δίνουν τους etext και end που αποτελούν σημαντική βοήθεια στο να βρεθεί η αρχή αλλά και το τέλος του data segment.
- Είναι εύκολο να γίνει προσπέλαση του top stack με τον esp register, το bottom φαίνεται να είναι «trickier».
- Η C δεν παρέχει κάποια συνάρτηση που να επιστρέφει τα variables που έχουν γίνει allocated στο stack.

Υλοποίηση

Αρχικά υλοποιήσαμε τη heapAlloc(), ένα αρκετά challenging κομμάτι καθότι ο allocator είναι πολύ σημαντικός. Οποτεδήποτε ο χρήστης ζητάει μνήμη δημιουργούμε ένα memory chunk στο οποίο αποθηκεύουμε την πρώτη διεύθυνση μνήμης που δίνεται στον χρήστη και το μέγεθος της. Δημιουργούμε δύο λίστες την ChunkAllocated και ChunkFreed. Σε αυτές «κρατάμε» τον αριθμό των chunks μνήμης που υπάρχουν σε κάθε λίστα και έναν πίνακα που αντιπροσωπεύει το heap.

Αρχικός στόχος ήταν οι δύο λίστες να υλοποιούνται με συνδεδεμένες λίστες και το struct να αποθηκεύει τον αριθμό των chunk μνήμης και τον pointer στο head της λίστας. Παρόλα αυτά, αντιμετωπίζαμε αρκετά προβλήματα και κατασπαταλήθηκε χρόνος στην προσπάθεια μας να κατανοήσουμε αν τα λάθη προκύπτουν από λανθασμένη υλοποίηση του allocator ή/και της συνδεδεμένης λίστας. Επομένως, χρησιμοποιήσαμε πίνακες στους οποίους αποθηκεύονται τα chunks.

Αυτή η απόφαση είχε επιπτώσεις, αν το μέγεθος του πίνακα (CHUNK_LIST_CAP) είναι μικρό τότε γεμίζει το heap ενώ υπάρχει διαθέσιμη μνήμη σε bytes για να παρασχεθεί στον χρήστη. Στην περίπτωση που το μέγεθος του πίνακα ισούται με τον αριθμό των bytes , το πρόγραμμα δεσμεύει υπερβολικά πολύ μνήμη (παραπάνω από όση χρειάζεται). Εν κατακλείδι, η μνήμη σε κάθε λίστα θα είναι περίπου το 1/4 του heap και θα εξηγήσουμε παρακάτω γιατί λάβαμε αυτή την απόφαση.

Άλλη μια παραδοχή, είναι ότι το heap θα έχει μέγεθος ίσο με μια virtual page των 4096 bytes.

Οποτεδήποτε ακούμε την λέξη μνήμη, όλοι σκεφτόμαστε έναν απλό πίνακα με bytes. Τόσο στην C , όσο και σε άλλες γλώσσες το char* θεωρείται ως block of memory. Οι άνθρωποι τείνουμε να θεωρούμε την μνήμη όπως το παράδειγμα παρακάτω:

Data																
Address	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F

Εν αντιθέσει, οι processors των υπολογιστών δεν διαβάζουν και δεν γράφουν στην μνήμη σε ενός-byte chunks μέγεθος, αλλά η προσπέλαση της μνήμης πραγματοποιείται σε δύο-, τέσσερα-, οχτώ-, δεκάξι ή και 32-byte chunks.

Data																	
Address	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	

Πως όμως επηρεάζεται ο processor από το memory access granuality; Αρχικά, διαβάζονται τέσσερα byte από τη διεύθυνση 0 του register.Στην συνέχεια διαβάζονται τέσσερα byte από τη διεύθυνση 1 στον ίδιο καταχωρητή.

Λόγω των παραπάνω, θα διαβάζουμε την μνήμη ανά 8 bytes καθώς στους 64-bit processorσ που έχουν οι υπολογιστές μας ένα word είναι ίσο με 8 bytes. Στην heapAlloc(), θέλοντας να δούμε οποιοδήποτε αριθμός bytes σε τι αριθμό word

αντιστοιχεί και παίρνοντας υπόψη το overflow. Αυτό γίνεται με στόχο να μπορούμε να διαβάσουμε την μνήμη ανά 8-bytes και να αποφύγουμε την προσπέλαση της μνήμης με sliding window με n=8.

Γιατί λοιπόν το capacity του πίνακα της κάθε μνήμης πιστεύουμε ότι θα είναι αρκετό; Αρχικά σπάνια όταν προγραμματίζουμε ζητάμε η μνήμη που θα μας είναι διαθέσιμη να μας δοθεί ένα-ένα byte. Η δυναμική μνήμη που χρησιμοποιούμε από την εμπειρία μας μέχρι τώρα και από ότι πιστεύουμε ότι και άλλοι προγραμματιστές χειρίζονται συνήθως, είναι αρκετά bytes. Υπενθυμίζουμε ότι το εν λόγω πρόβλημα προκύπτει γιατί δεν ελέγχουμε την δεσμευμένη και την ελεύθερη μνήμη με συνδεδεμένες λίστες αλλά με πίνακα.

Η heapAlloc() τελικά είναι best fit (και όχι first fit όπως σχεδιαζόταν αρχικά) κάτι που βοήθησε το γεγονός ότι είναι υλοποιημένο με πίνακα αντί για λίστα. Χρησιμοποιώντας την συνάρτηση ChunkListRemove() αφαιρούμε το best fit κομμάτι από την freedList, με την ChunkListInsert() το προσθέτουμε στην AllocatedList() και ενόσω περισσεύει κάποιο κομμάτι μνήμης από το best fit chunk το προσθέτουμε στην freedList με την ChunkListInsert().

Αποφασίσαμε ότι η heapCollect() θα πραγματοποιείται όταν δεν υπάρχει διαθέσιμο κομμάτι μνήμης που να είναι μεγαλύτερο από το μέγεθος που ζητείται από τον χρήστη ή όταν το capacity του πίνακα της λίστας φτάνει το μέγιστο μέγεθος. Η παρούσα συνθήκη είναι ένα υπαρκτό μειονέκτημα λόγω της χρήσης πίνακα αντί για λίστα.

Για τον Garbage Collector θα δείτε ότι στην υλοποίηση μας σκανάρουμε μόνο το stack από όλα τα root pointers και όχι το data segment. Αυτό συμβαίνει γιατί δεν έχουμε ολοκληρώσει με επιτυχία την υλοποίηση του Garbage Collector.

Για να γνωρίζουμε την αρχή και το τέλος του stack χρησιμοποιούμε την εξής built-in function **__builtin_return_address** και όχι assembly όπως είχαμε εκτιμήσει πριν ξεκινήσουμε την υλοποίηση μας.

Στην συνέχεια ψάχνουμε ανά 8 bytes για να βρούμε τιμές οι οποίες ανήκουν στο semispace heap που χρησιμοποιούμε την εκάστοτε φορά με σκοπό να βρεθούν τα stack roots. Με το που βρεθεί το root ακολουθούμε τον αλγόριθμο του Cheney υλοποιώντας τον με μια δικιά μας απλοποιημένη εκδοχή.

Η υλοποίηση μας αντιμετωπίζει προβλήματα όταν προσπαθούν να αντιγραφούν τα κομμάτια μνήμης από το ένα heap στο άλλο. Δεν αντιγράφεται όλο το μέρος των live αντικειμένων και κάποιοι από τους pointers δεν δείχνουν στην σωστή θέση μνήμης.

Επόμενα βήματα εφόσον είχαμε χρόνο:

- Debugging του heapCollect().
- Parsing του data segment. (Εύκολο εφόσον είχαμε έτοιμο το stack)
- Αντικατάσταση των πινάκων που ελέγχουν την δεσμευμένη και την ελεύθερη μνήμη με συνδεδεμένες λίστες.

- Δέσμευση μνήμης για το heap μας δυναμικά με system call αντί για στατική δέσμευση από το data segment.
- Evaluation.

Οδηγίες για τρέξετε την υλοποίηση

Για να τρέξετε τον κώδικα παρακαλούμε εγκαταστήστε:

- MinGW Distro για λειτουργικό Windows

Για να δείτε τον κώδικα συνιστάται:

- VSCode
- Μέσω του VSCode(συνιστάται να εκατασταθεί το extension "C/C++" της Microsoft.

Αφού κάνετε clone το παρόν repository:

git clone https://github.com/categerjsy/HY446-Final-Repo-CGC.git

Ανοίξτε τον terminal σας στο εσωτερίκο του cloned φακέλου.

Παρακαλούμε κάντε compile τα files με την παρακάτω εντολή:

gcc -o heap main.c heap.c

Για να τρέξετε το πρόγραμμα αρκεί να δώσετε την εντολή: **./heap**

Evaluation

Πιστεύουμε ότι νόημα θα είχε να γίνει το evaluation σε προγράμματα με data structures. Ένας τρόπος θα ήταν να γράψουμε ένα πρόγραμμα που να εκτελεί τις ίδιες διαδικασίες με τη υλοποίηση του Garbage Collector και με απλή χρήση της <stdlib.h> βιβλιοθήκης.

Επιπρόσθετα αν ο χρόνος το επέτρεπε σχεδιάζαμε να βρούμε open source προγράμματα που να έχουν memory leaks και να τα ξαναγράψουμε χρησιμοποιώντας την υλοποίηση μας.

Δυστυχώς δεν προλάβαμε να πραγματοποιήσουμε ούτε τον αρχικό μας στόχο στο evaluation.

Πηγές

i. http://www.cs.cornell.edu/courses/cs312/2003fa/lectures/sec24.htm
ii. https://stackoverflow.com/questions/40711589/why-does-c-not-require-a-garbage-collector
iii. https://maplant.com/gc.html
iv. https://developer.ibm.com/articles/pa-dalign/
v. https://www.javatpoint.com/structure-padding-in-c
vi. https://gcc.gnu.org/onlinedocs/gcc/Return-Address.html
vii. https://www.cs.princeton.edu/courses/archive/spring16/cos320/lectures/13-GC.pdf