Čiščenje pomnilnika

Matija Čufar

Uvod

V svoji seminarski nalogi bom predstavil problem čiščenja pomnilnika in različne načine na katere ga rešujemo, sploh tiste, ki se danes še uporabljajo. Največ pozornosti bom posvetil avtomatskemu čiščenju pomnilnika (garbage collection). Začel bom z opisom problema in kratko zgodovino, nato predstavil najpogosteje uporabljene algoritme, za konec pa še podal nekaj konkretnih primerov iz programskih jezikov, ki so se mi zdeli pomembni ali zanimivi.
Ker za vse termine ne poznam slovenskih izrazov, bom nekatere izraze uporabljal v angleščini.

Opis problema

Računalniki pri svojem delu uporabljajo pomnilnik. Praktično vsi sodobni programski jeziki pomnilnik razdelijo na sklad (stack) in kopico (heap).
Delo s skladom je preprosto, saj je urejen in nanj podatke lahko nalagamo z ukazom `PUSH` in jih iz njega jemljemo z ukazom `POP`, njegov problem pa je, da je ponavadi v primerjavi s kopico majhen, in da so stvari, ki so naložene nanj navadno vidne samo iz enega dela podprograma.
Delo s kopico je bolj zapleteno, saj ta ni strukturirana, tako da mora program za vsak podatek vedeti kje se nahaja. Problem nastane tudi pri tem, da pri delu s kopico potrebujemo mehanizem, s katerim zagotovimo, da iz nje brišemo nepotrebne podatke. Če tega ne bi počeli bi nam hitro zmanjkalo pomnilnika.

Zgodovina

V zgodnjih letih računalništva, v štiridestih letih prejšnjega stoletja, so računalnike programirali v strojni kodi. Ti računalniki so imeli zelo malo pomnilnika, tako da se je za delo z njim v glavnem uporabljala statična alokacija. Pri statični alokaciji se deli uporabljenega pomnilnika določijo že med prevajanjem, oz. v primeru strojne kode med pisanjem programa.
Kasneje, v sredini petdesetih let, so na plan prišli prvi višje-nivojski programski jeziki, a so še vedno v glaven uporabljali statično alokacijo, nekateri, npr. ALGOL, pa so za delo s pomnilnikom uporabljali izključno sklad.
Ideja avtomatskega dela s pomnilnikom se je pojavila leta 1959, ko ga je John McCarthy izumil prvi garbage collector, z namenom, da se uporablja v programskem jezik LISP.
LISP je takrat uporabljal preprost algoritem, ki je zaznal, da je prostora na pomnilniku zmanjkalo, poiskal vse objekte, ki programu niso več dosegljivi in jih izbrisal. To je bilo zelo zamudno, zato se avtomatsko čiščenje pomnilnika pri drugih jezik ni prijelo dolgo časa.
Zares popularno je postalo šele pred kratkim, z izidom programskega jezika

Java, leta 1995, danes pa ga uporabljajo skoraj vsi programski jeziki, z redkimi izjemami, kot sta C in C++.

Ročno delo s pomnilnikom

Z ročnim delom s pomnilnikom se danes v glavnem srečamo v programskih jezikih C in C++, uporabljajo ga tudi nekateri novejši jeziki namenjenu sistemskemu programiranju, kot je npr. Rust, v nekaterih jezikih, kot je npr. D lahko garbage collection za kritične dele kode izklopimo.
Ročno delo pomnilnika se v glavnem izvaja na dva načina:

- Eksplicitni klici funkcij za alokacijo in dealokacijo pomnilnika. Ta način se uporablja v programskem jeziku C s klici funkcij `malloc` in `free`.
- Smart pointerji - pomnilnik se počisti, ko kazalec na nek podatek izgine iz scopa. Pri tem je pomembno, da imamo na vsak podatek samo en kazalec. To se uporablja v C++11 in v Rustu. C++11 in Rust uporabljata tudi reference counting, ki je opisan v naslednjem poglavju. Ta način je varnejši kot eksplicitni klici funkcij `malloc` in `free`.

Ročno delo s pomnilnikom je uporabno v primerih, ko imamo pomnilniški prostor omejen, ali pa ko potrebujemo zelo predvidljivo prabo pomnilnika in časa izvajanja. Kljub popularnem prepričanju med programerji, je avtomatsko čiščenje pomnilnika pogosto hitrejše, pred vsem pa bolj varno kot ročno. Nekaj primerov nevarnosti ročnega dela s pomnilnikom:

- Memory leak - Pomnilnik smo rezervirali, a pozabili sprostiti. Zaradi tega program s časom zasede vedno več pomnilnika, dokler ga ne zmanjka. Avtomatsko upravljanje s pomnilnikom tega programa ne odpravai nujno v celoti.
- Dangling pointer - Pomnilnik sprostimo, a imamo še vedno kazalec na sproščen del. To ima lahko nepredvidljive posledice.
- Double free - Že sproščen kos pomnilnika poskusimo sprostiti še enkrat. To povzroči segmentation fault.

Reference counting

Reference counting ali štetje referenc je preprost način čiščenja spomina. Deluje tako, da v vsak objekt v pomnilniku poleg njegovih podatkov shrani še število referenc nanj. Ko to število doseže 0, to pomeni, da je objekt programski kodi nedostopen, zato ga lahko počisti.
Njegove prednosti so:

- Preprosta implementacija - najpreprostejša različica te metode ne potrebuje podpore prevajalnika, npr. v C++ je implementirana kot razred, `smart pointer`.
- Predvidljivo delovanje - vemo, da bo spomin počiščen takoj, ko bo objekt postal nedostpoen.

Kjub preprostosti ima metoda nekaj slabih strani.

- Ciklične reference - pri njih se zgodi, da števec referenc objekta nikoli ne pade na 0, čeprav je programski kodi nedostopen. To se lahko zgodi npr. pri dvojno povezanih seznamih ali pr grafih, ki dovoljujejo cikle. Problem se rešuje z dodatnim algoritmom, ki poskrbi za detekcijo ciklov, ali pa z uvedbo t.i. weak pointerjev - kazalcev, ki ne ob stvaritvi ne povečajo števca.
- Števci so ponavadi fiksne dolžine, kar efektivno omejuje število dovoljenih referenc na en objekt.
- Dodatno delo - Pri vsakem nastanku ali izbrisu reference je potrebno spremeniti vrednost števca. To pokvari prostorsko lokalnost in je sploh problematično pri sočasnih programih, saj je tam za vsako posodobitev potrebno niti med seboj uskladiti. Problem lahko nastane tudi pri čiščenju velikih objektov, ki lahko čas izvajanja programa za nekaj časa popolnoma ustavijo.
- Dolgotrajno čiščenje velikih objektov - Objekt moramo iz pomnilnika počistiti v celoti, kar je lahko pri velikih objektih zelo zamudno.

Poleg osnovnega algoritma obstaja še nekaj izboljšav, kot je npr. deferred reference counting.
Deferred reference counting ne poveča števca referenc objektom, kazalci na katere so shranjeni na skladu, saj vemo, da ti ne bodo dolgo v uporabi. To pa prinaša dodatne komplikacije s tem, da v takem primeru objektov s števcem z vrednostjo 0 ne smemo počistiti takoj, saj bi lahko na skladu obstajale reference do njih. Za to tehniko potrebujemo podporo prevajalnika. To tehniko med drugim uporablja programski jezik Smalltalk.

Kljub slabostim se metoda pogosto uporablja, sploh v povezavi s tracing garbage collectorjem, ki skrbi za ciklične strukture.

Tracing garbage collection

Tracing garbage collection je skupina algoritmov, ki deluje tako, da se 'sprehodi' po vseh referencah in si zapomne kateri objekti so v uporabi. Nekateri viri besedo garbage collection uporabljajo kot sinonim za tracing garbage collection, reference counting pa štejejo posebej.
Poskusil bom našteti čimveč variant na algoritem, bi pa rad opozoril, da jih je

zelo veliko, zato bom verjetno kakega izpustil.
V vsakem podpoglavju bom predstavil različice, ki rešujejo isti problem. Izpostavil bi še to, da različni viri uporabljajo različne izraze za različne metode.

Stop-the-world, Incremental, Concurrent, Parallel

- Izraz stop-the-world pomeni, da garbage collector, ko enkrat začne s čiščenjem spomina, ne more biti prekinjen. To ima posledico, da se lahko izvajanje programa za nekaj časa ustavi, kar je problematično pri real-time sistemih.
- Incremental garbage collector deluje podobno, kot stop-the-world, le da lahko svoje izvajanje prekinja in prepušča nadzor programu. Ponavadi je implementiran tako, da se izvaja na fiksnih intervalih za fiksno količino časa.
- Concurrent garbage collection je podoben kot incremental, le da ima bolj pametno izbiro časov, med katerimi se izvaja, npr. lahko izvede veliko dela med tem, ko čaka na uporabnikov odziv, ali ko se izvaja neka zunanja koda. Teče sočasno s programom, ni pa nujno, da paralelno.
- Parallel garbage collector teče na svoji niti, tako, da lahko svoje delo opravlja vzporedno z izvajanjem programa. Parallel garbage collector je lahko concurrent, incremental ali stop-the-world.

Precise, Conservative

- Precise garbage collector pri iskanju objektov sledi le tistim poljem struktur, ki so definirane kot reference. To se uporablja v praktično vseh jezikih, saj po navadi ne dovoljujejo aritmetike s kazalci.
- Conservative garbage collector kot reference tretira vsa polja v objektih, tudi npr. integerje, saj bi se lahko za njimi skrivale reference. To pride v poštev pri jezikih, kot je C, ki dovoljujejo cast kazalca na kak drug podatkovni tip. Problem pri temu je to, da lahko včasih najde lažne reference, a to niti ni tako velik problem, saj je verjetnost da se to zgodi zelo majhna, izguba spomina pa zaradi tega ni velika.

Non-moving, Moving

- Non-moving garbage collector med izvajanjem programa objektov, ki so že v spominu ne prestavlja. To je preprostejše implementirati in deluje hitreje, a zaradi tega lahko pride do zunanje fragmentacije. Kot non-moving lahko štejemo tudi reference counting in večino primerov ročnega upravljanja s pomnilnikom.
- Moving garbage collector med izvajanjem objekte prestavlja. Ponavadi jih poskuša zgostiti v bloke v pomnilniku.

Operacije Mark, Sweep, Compact, Copy

- Operacija Mark preišče vse objekte v pomnilniku in označi tiste, ki jih lahko doseže. Iskanje začne v vseh referencah, ki so direktno vidne programu, kot so spremenljivke na skladu in statične spremenljivke. Časovna zahtevnost operacije je ponavadi linearna velikosti žive množice.
- Operacija Sweep preišče celotno kopico in reciklira objekte, ki niso označeni za žive. Časnovna zahtevnost je ponavadi linearna velikosti kopice.
- Operacija Compact vse žive objekte premakne tako, da med njimi ni praznega prostora. Časovna zahtevnost je ponavadi linearna velikosti žive množice.
- Operacija Copy preišče množico podobno, kot mark, le da jih namesto označevanja prestavi v drugo kopico. Stara kopica se po operaciji smatra za prazno. Čas trajanja je linearen velikosti žive množice.

Konkretni algoritmi za tracing garbage collection

- Mark-sweep deluje v dveh fazah. V fazi mark se preišče vse žive objekte in označi tiste, ki so še dostopni, v fazi sweep pa počisti nedostopne.
- Mark-sweep-compact deluje podobno, kot mark-sweep, le da v tretji fazi objekte prestavi tako, da ne pride do fragmentacije.
- Mark-compact deluje podobno, kot mark-sweep-compact, le da operacijo sweep preskoči, operacijo compact pa izvede v drug del kopice. Je hitrejši, kot mark-sweep-compact, a potrebuje večjo kopico, vsaj dvakrat večjo od velikosti žive množice.
- Mark-don't-sweep deluje tako, da vse objekte tretira kot žive, dokler prostor na pomnilniku ne poide. Ko se to zgodi, vse objekte označi kot mrtve, na to pa izvede fazo mark, s katero spet označi žive. Vsi objekti, ki v tej fazi niso bili označeni, se tretirajo kot prost pomnilnik.
- Generational garbage collector uporablja več kopic, ponavadi dve, eno za mlade objekte in eno za stare. Temeli na spoznanju, da 10-20% objektov živi 80-90% časa, zato lahko uporablja različne strategije za mlade in stare objekte. Ponavadi z redkejšim čiščenjem starejših objektov prihrani veliko procesorskega časa. Ker stari objekti redko (v nekaterih funkcijskih jezikih celo nikoli) kažejo na stare, se lahko take refernce upravlja posebej, kar dodatno pohitri delovanje. Ta tehnika se imenuje remembered set.

Konkretni primeri programskih jezikov

Java

Javin JVM uporablja generational garbage collection. Za mlajšo generacijo uporablja copying stop-the-world collector, ki vse objekte, ki preživijo prestavi v

starejšo generacijo, celoten prostor mlajše pa obravnava kot prost. Za starejšo generacijo se uporablja concurrent mark-sweep algoritem. Obstajajo tudi drugačne implementacije JVM-ja, ki uporabljajo različne strategije. Tu sem opisal delovanje OpenJDKja in Oracle JDKja.

Python

Python uporablja reference counting v povezavi z algoritmom, ki šteje razliko med številom alokacij in dealokacij. Ko to število preseže neko mejo, se požene algoritem podoben mark-and-sweep, ki išče cikle in že od začetka ignorira strukture, ki ne morejo biti ciklične (kot npr. `string`). Če najde cikel, vsem objektom v ciklu zmanjša števec referenc za 1. Če ta doseže 0, lahko počisti ciklično strukturo.

C++

C++ podpira veliko tehnik za ročno urejanje spomina, trenutno, v C++11, pa je najbolj priporočena uporaba unique pointerjev, ki se počistijo, ko grejo iz scopa in smart pointerjev, ki uporabljajo reference counting. Podoben model upravljanja s pomnilnikom uporablja Rust.

Haskell

Haskell uporablja parallel generational garbage collection. Ker so vsi objekti v Haskellu nespremenljivi to močno olajša delo z generacijami, saj stari objekti ne morejo imeti referenc na nove. Zato uporablja majhen (~512kb) 'nursery', v katerega grejo novi objekti. Ko se ta napolne, se vse objekte iz njega, ki so še dostopni prekopira med stare. Pri iskanju dostopnosti lahko ignorira stare objekte.

Mercury

Mercury - deklarativen jezik, ki spominja na mešanico Haskella in Prologa, vse delo garbage collectorja opravi že med prevajanjem. Natančnega delovanja ne poznam, saj je edina referenca nanj cca. 380 strani dolga knjiga, ki formalno dokaže pravilnost njegovega delovanja[1]. Zame je to prevelik zalogaj, je pa zanimivo vedeti, da gre razvoj tudi v to smer.

[1] https://www.mercurylang.org/documentation/papers/CW2004_03_mazur.pdf

Viri

Med nabiranjem informacij, si žal nisem zapisoval vseh virov. Pregledal sem kar nekaj spletnih strani. Pri pisanj mi je bila v veliko pomoč spletna stran http://www.memorymanagement.org/. Uporabno je bilo tudi predavanje na naslovu https://www.youtube.com/watch?v=LJC_2agoLuE.

Za konkretne primere programskih jezikov sem večinoma podatke o njih našel na njihovih uradnih spletnih straneh.